Response Letter for Submission "Maximum k-Plex Search: An Alternated Reduction-and-Bound Method" (Paper ID: 308)

We would like to sincerely thank the reviewers and meta-reviewer for their valuable comments and suggestions. We have revised our paper accordingly (the changes are highlighted in blue). Below please find our detailed responses to Reviewers' comments.

Detailed Comments to Reviewer #1

W1/D1. *The paper's novelty could be improved. The new alternate reduction-and-bound method, AltRB, just is an incremental optimization of existing methods, which limits its novelty. Furthermore, the relationship between AltRB and the two preprocessing optimizations is not well established.*

Response to W1/D1: We first clarify on the main contributions of `AltRB`. First, `AltRB` follows a novel branch-and-bound framework, which differs from the one that has been adopted by all recent proposed methods. We have discussed the differences in the last paragraph of Section 3. Second, we note that the newly proposed `AltRB` can be potentially adapted to solving other graph mining problems, including finding the maximum clique and finding the maximum defective clique, to achieve better piratical performance.

Then, we clarify that the two preprocessing optimizations, namely `CF-CTCP` and `KPHeuris`, are *orthogonal* to the branch-and-bound methods including our `AltRB`. In specific, as shown in Algorithm 1, `KPHeuris` and `CF-CTCP` are conducted at Stage-I for refining the input graph to a smaller one and obtaining a tighter lower bound, which are prior to applying the branch-and-bound method `AltRB` at Stage-II. Besides, we note that with `KPHeuris` and/or `CF-CTCP`, our `AltRB` would achieve better practical performance, as verified in Table 5 of our experimental studies. We have included the above discussion in Section 5 of the revised paper.

W2/D2. *The method for partitioning the branch requires more discussion. The paper's core idea is branch partitioning, which needs more thorough discussion. Currently, the authors present only a greedy partitioning method. It's crucial to explore whether different partitioning methods yield varying performances and how the partitioning results impact overall performance. Both theoretical and experimental analyses are needed to address these questions.*

Response to W2/D2: First, from the theoretical perspective, we note that finding the "best" partition towards pruning the largest number of vertices from the candidate set C is hard to solve since (1) the newly proposed reduction rules **RR1** and **RR2** are non-trivial and depend on the largest k-plex S^* seen so far, and (2) the number of possible binary partitions is exponentially large, i.e., $O(2^{|S|+|C|})$. As a result, it is theoretically untraceable how different partition methods affect the time complexity of `kPEX`. Hence, we propose to explore heuristic partitions towards achieving better practical performance. In addition, we remark that with our proposed greedy partition method `Partition`, `AltRB` can prune more vertices from C compared to `SeqRB` in theory, as shown in Equation (9), while other partition methods may not. We have included the above discussion in Section 4.2 of the revised paper.

Besides, from the experimental perspective, we explore three different partition methods, namely (1) `Partition`: Algorithm 3, (2) `dp`: the degree-based partition which sorts the vertices in S (resp. C) based on their degrees and partitions S (resp. C) into two subsets with the same size, and (3) `rp`: the random partition which randomly partitions S (resp. C) into two subsets. Note that all of them can be done in polynomial time. We study the effect of various partition methods on the running time of `kPEX` and report the results in Table 7. We observe that our `Partition` outperforms others by achieving at least 10x speedup on 11 out of 30 graphs, which demonstrates its practical effectiveness. This is also consistent with the theoretical analysis given in Equation (9) that `AltRB` with `Partition` further narrows down the search space in theory.

W3/D3. *Some algorithm details need further explanation. Some aspects of the algorithm require clearer explanations. For instance, Line 7 in Algorithm 1 appears problematic. After Line 2, a reduced graph G is obtained. Since Lines 3-6 do not modify G, it's unclear why G would change in Line 7. The authors might have omitted steps, such as removing vertex v from G, which should be clarified.*

Response to W3/D3: We first clarify that graph G would be refined via the procedure of `CF-CTCP` in Line 7 (now Line 8 in the revision) since (1) a tighter lower bound $|S^*|$ could be obtained in Line 6 (note that `CF-CTCP` relies on the lower bound and could potentially prune more vertices from G as the lower bound becomes larger) and (2) one vertex v can be removed from G (as all k-plexes containing v have been found at Line 6). We have provided the above explanation and the details of invoking `CF-CTCP` for refining the graph G in Line 2 and Lines 7-8 of Algorithm 1 in the revision.

Detailed Comments to Reviewer #2

W1/D4. *While the authors acknowledge instances where kPEX underperforms compared to SOTA algorithms, their explanation relying on heuristics could be more detailed. A more detailed analysis of these cases is sorely needed, perhaps including characteristics of the problematic graphs?*

Response to W1/D4: We note that our `kPEX` runs slightly slower than the SOTA algorithm `kPlexT` on G23 as shown in Table 3. The possible reasons are as follows. First, we observe that in G23, the size of the maximum 5-plex is 11. Both `kPEX` and `kPlexT` find the maximum 5-plex after the pre-processing procedure, as verified by *lb* (which equals to 11 exactly) in Table 6. Hence, they have the running time dominated by that of conducting the pre-processing method. Second, our pre-processing methods (i.e., `CF-CTCP` and `KPHeuris`) have the time complexity of $O(nm)$, and thus they cost more than the existing one which has the time complexity of $O(\delta m)$. We have included the above analysis in Section 6.1.

W2/D5. *Theoretical and experimental analysis of space (memory) usage is not presented and is much needed. Given the potential for large graph processing to be memory-intensive, understanding the algorithm's space requirements is crucial. What is theoretical space complexity bounds for kPEX and its components? Authors should provide empirical measurements of memory usage across different graph sizes and k values, also compared to SOTA approaches.*

Response to W2/D5: We have provided the theoretical analysis of space complexity (Section 4.3) and conducted additional experiments (Section 6.1) in the revision. Below please find some details.

We note that our kPEX has the space complexity of $O(m+n)$, which is the same as that of the SOTA algorithm kPlexT. In specific, in Stage-I, CF-CTCP needs to maintain two sets Q_v and Q_e, which have the size of $O(n+m)$ and thus dominate the space complexity. In Stage-II, BRB_Rec recursively maintains three global data structures, namely g, S, and C, for each branch, which dominate the space complexity. Here, graph g is obtained at Line 5 of Algorithm 1 and thus has the size bounded by $O(n+m)$. S and C are two disjoint vertex sets and have the size bounded by $O(n)$ clearly.

Besides, we measure the peak memory usage of kPEX and baselines, and report the results in Table 4. We observe that our kPEX achieves comparable memory usage compared to the SOTA algorithm kPlexT, which is also aligned with the theoretical analysis.

Detailed Comments to Reviewer #3

W1/D2. *The experiments show that the proposed methods outperform the baselines significantly. However, in this paper, KPLEX is remarkably better than kPlexT, which contradicts to the results in the original paper of kPlexT (where kPlexT is claimed to the state of the art). Please confirm the experimental results and provide a justification.*

Response to W1/D2: We clarify that the differences in performance are due to the different nature of the studied problems. Specifically, our paper focuses on the problem of finding the maximum k-plex *with size at least* $2k-1$, while kPlexT in [12] studies the problem of finding the maximum k-plex *without any size constraint*. As mentioned in Section 2, following the previous studies [11, 53], we study the maximum k-plex problem with size constraint since (1) a k-plex with size at most $2k-2$ is less informative in practice, and (2) a k-plex of size at least $2k-1$ is more cohesive as its diameter is at most 2. In our experiments, for fair comparison, we run all algorithms with an initial lower bound of $2k-2$.

In the technical report of our paper [29], we further provide experimental results on the problem of finding maximum k-plex without any size constraint. In particular, we adapt kPEX by removing the restriction of $|S^*| \geq 2k-1$ for this problem. The results are reported in Table 8 in Appendix B.1. We observe that both kPlexT and kPlexS outperform KPLEX, as also observed in [12]; and our kPEX still achieves the best performance.

W2/D3. *As given in Definition 2.2, k-plex is strongly connected k-core/k-truss. Intuitively, the algorithms for k-core/k-truss (with some simple modifications if needed) can be easily applied to k-plex. However, such a comparison/discussion is missing.*

Response to W2/D3: We clarify that modifying algorithms for k-core/k-truss to solve the maximum k-plex search problem is not easy or practical. This is because the inherent natures of k-plex and k-core/k-truss are fundamentally different: the k-plex focuses on the number of *disconnections* to other vertices for each vertex, while k-core/k-truss (roughly) consider the number of *connections* to other vertices for each vertex/edge. Consequently, for example, the maximum k-plex search problem is NP-hard for any fixed k [3], while the maximum k-core/k-truss search problems can be polynomially solved in $O(m)$ [4] and $O(m\delta)$ [51], respectively. Thus, existing studies and ours mainly apply algorithms for k-core/k-truss to conduct reductions (e.g., in CF-CTCP) instead of exactly solving the maximum k-plex search problem.

W3/D4. *In [12], there is a clear comparison of time complexity of different algorithms. It is better to provide such a comparison.*

Response to W3/D4: Thanks for the suggestion. We have compared the time complexity of different algorithms in the related work. We elaborate on the details as below.

Quite a few algorithms are proposed towards achieving better worst-case time complexity. In specific, the worst-case time complexity has been improved from $O^*(2^n)$, to $O^*(\beta_k^n)$ [56], to $O^*((k+1)^{\delta+k-|S^*|})$ [53] and $O^*(\gamma_k^\delta)$ [12], where O^* suppresses the polynomial factors and $\gamma_k < \beta_k < 2$. Among them, KPLEX [53] and kPlexT [12] achieve the best. We note that (1) previous state-of-the-art algorithms are all branch-and-bound methods and the improvements of time complexity are attributed to different branching strategies, which are orthogonal to the techniques (e.g., reduction rules and the pre-processing methods) proposed in this paper; and (2) the practical performance of different algorithms mainly depends on the later. Hence, by adopting the branching strategies used in KPLEX and kPlexT, our kPEX can achieve the same time complexity of $O^*((k+1)^{\delta+k-|S^*|})$ and $O^*(\gamma_k^\delta)$, respectively (note that the analysis depends on the branching strategy only and thus is the same as KPLEX and kPlexT respectively). Besides, we also compare the empirical performance of different branching strategies and find that our kPEX with different branching strategies achieves comparable performance (details can be found in the technical report [29]).

Maximum k-Plex Search: An Alternated Reduction-and-Bound Method

Shuohao Gao
Harbin Institute of Technology, Shenzhen
China
200111201@stu.hit.edu.cn

Kaiqiang Yu
Nanyang Technological University
Singapore
kaiqiang002@e.ntu.edu.sg

Shengxin Liu
Harbin Institute of Technology, Shenzhen
China
sxliu@hit.edu.cn

Cheng Long
Nanyang Technological University
Singapore
c.long@ntu.edu.sg

ABSTRACT

k-plexes relax cliques by allowing each vertex to disconnect to at most k vertices. Finding a maximum k-plex in a graph is a fundamental operator in graph mining and has been receiving significant attention from various domains. The state-of-the-art algorithms all adopt the branch-reduction-and-bound (BRB) framework where a key step, called *reduction-and-bound* (RB), is used for narrowing down the search space. A common practice of RB in existing works is `SeqRB`, which *sequentially* conducts the reduction process followed by the bounding process *once* at a branch. However, these algorithms suffer from the efficiency issues. In this paper, we propose a new *alternated reduction-and-bound* method `AltRB` for conducting RB. `AltRB` first partitions a branch into two parts and then *alternatively* and *iteratively* conducts the reduction process and the bounding process at each part of a branch. With newly-designed reduction rules and bounding methods, `AltRB` is superior to `SeqRB` in effectively narrowing down the search space *in both theory and practice*. Further, to boost the performance of BRB algorithms, we develop efficient and effective pre-processing methods which reduce the size of the input graph and heuristically compute a large k-plex as the lower bound. We conduct extensive experiments on 664 real and synthetic graphs. The experimental results show that our proposed algorithm `kPEX` with `AltRB` and novel pre-processing techniques runs up to two orders of magnitude faster and solves more instances than state-of-the-art algorithms.

PVLDB Reference Format:
Shuohao Gao, Kaiqiang Yu, Shengxin Liu, and Cheng Long. Maximum k-Plex Search: An Alternated Reduction-and-Bound Method. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ShuohaoGao/kPEX.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1 INTRODUCTION

The graph model serves as a versatile tool for abstracting numerous real-world data which captures relationships between diverse entities in social networks, biological networks, publication networks, and so on. Cohesive subgraph mining is one of the central topics in graph analysis and data mining where the objective is to mine those *dense or cohesive* subgraphs that normally bring valuable insights for analysis [10, 25, 26, 34, 39]. For example, cohesive subgraph mining has been used to detect a terrorist cell in social networks [38], to identify protein complexes in biological networks [62], and to find a group of research collaborators in publication networks [32].

The *clique* is arguably the most well-known cohesive subgraph where every pair of distinct vertices is connected by an edge. In the literature, the study of efficient algorithms for extracting the maximum clique or enumerating maximal cliques is extensive, e.g., [7, 8, 18, 24, 44, 45, 49, 50]. Nevertheless, clique, being a tightly interconnected subgraph, is over-restrictive, which limits its practical usefulness. To circumvent this issue, relaxations of clique have been proposed and studied in the literature, such as k-plex [48], k-core [47], quasi-clique [31, 58], and k-defective clique [9, 21]. In particular, k-plex relaxes clique by allowing each vertex to disconnect to at most k vertices (including the vertex itself). It is clear that 1-plex corresponds to clique. The research of cohesive subgraph mining in the context of k-plex has recently attracted increasing interests [11, 12, 22, 27, 35, 36, 53, 54, 56, 63].

In this paper, we study the *maximum k-plex search* problem which aims to search the k-plex with the largest number of vertices in the given graph. It is well-known that the maximum k-plex search problem is NP-hard for any fixed k [3]. Thus, existing studies and ours focus on designing practically efficient algorithms.

Existing algorithms. The state-of-the-art algorithms all (conceptually) adopt the *branch-reduction-and-bound* (BRB) framework [11, 12, 27, 35, 36, 53, 63]. The idea is to recursively solve the problem instance (or branch) by solving the subproblem instances (or sub-branches) produced via a process of *branching*. A branch denoted by (S, C) corresponds to a problem instance of finding the largest k-plex from the subgraph (of the input graph) induced by vertex set $S \cup C$, where the partial solution S corresponds to a k-plex and the candidate set C corresponds to the set of vertices used to expand the partial solution. We refer the search space of a branch to the set of possible k-plexes in the subgraph induced by $S \cup C$. At each branch, a key step, named *reduction-and-bound* (RB), is performed for narrowing down the search space. We note that existing studies all follow a sequential framework, called SeqRB, for implementing the RB step. Specifically, SeqRB *sequentially* conducts two processes *once*: 1) the reduction process shrinks the candidate set C by removing some unpromising vertices that cannot appear in the largest k-plex; and 2) the bounding process computes the upper bound of the size of the largest k-plex in the branch refined by the first step, which is used for pruning unnecessary branches (i.e., with the upper bound no larger than the largest k-plex seen so far). The rationale behind is that with some vertices being removed by the reduction process, the bounding process may obtain a smaller upper bound so as to prune more branches.

Existing studies focus solely on refining the reduction and bounding methods used in SeqRB while devoting little effort to improving the whole RB framework. We observe that, in SeqRB, the reduction process benefits the bounding process, but not the other way; thus, they are sequentially conducted only once. One interesting question is that: *Can we design a new RB framework where the reduction process and the bounding process can benefit each other*?

We note that recent studies [11, 12, 53, 63] boost the practical performance of BRB algorithms by devising pre-processing techniques. These techniques include 1) graph reduction algorithms [11, 63] for reducing the size of the input graph (among which the best one is CTCP [11]); and 2) heuristic algorithms [11, 12, 63] for computing an initial large k-plex used for the above reduction algorithms (among which the best ones are kPlex-Degen [11] and EGo-Degen [12]).

Our new methods. In this paper, we first propose a new framework, called *alternated reduction-and-bound* (AltRB), for conducting the RB step at a branch (S, C). AltRB differs from SeqRB mainly in the way of conducting the reduction process and the bounding process. Specifically, AltRB first partitions a branch into two parts (i.e., $S = S_L \cup S_R$ and $C = C_L \cup C_R$). With newly-designed reduction rules and upper bound computation methods on each part, *the bounding process on one part will benefit the reduction process on the other* (note that the reduction process still benefits the bounding process on the same part, which is the same as SeqRB). Thus, AltRB *alternatively* and *iteratively* conducts the reduction process and the bounding process at each part of a branch (e.g., bounding on $S_L \cup C_L \rightarrow$ reduction on $S_R \cup C_R \rightarrow$ bounding on $S_R \cup C_R \rightarrow$ reduction on $S_L \cup C_L \rightarrow$ ...). In this manner, the bounding process and the reduction process could mutually benefit from each other. We show that AltRB is superior to SeqRB in narrowing down the search space in both theory (as will be shown in Equation (9)) and practice (as will be shown in Table 3). We further design efficient pre-processing techniques for boosting the practical performance of BRB algorithms: 1) a new method CF-CTCP, which differs with CTCP in the way of conducting different reductions at each iteration, and 2) a heuristic algorithm KPHeuris that iteratively compute a large initial maximal k-plex.

With all the above newly-designed techniques, we develop a new BRB algorithm called kPEX, which runs up to two orders of magnitude faster and solves more instances than state-of-the-art algorithms kPlexT [12], kPlexS [11], KPLEX [53], and DiseMKP [35].

Our contributions. Our main contributions are as follows.

- We propose a new BRB algorithm called kPEX, which integrates the *alternated reduction-and-bound* method AltRB (Section 3). With our novel reduction and bounding methods, AltRB is superior to SeqRB in narrowing down the search space (Section 4).
- We design efficient pre-processing techniques for boosting the performance of BRB algorithms, namely a new method CF-CTCP for reducing the size of the input graph and a heuristic KPHeuris for computing a large initial k-plex (Section 5).
- We conduct extensive experiments on 664 graphs to verify the effectiveness and efficiency of our algorithms. Compared with state-of-the-art algorithms, our kPEX 1) solves most number of graph instances within the time limit and 2) runs up to two orders of magnitude faster than existing algorithms (Section 6).

2 PRELIMINARIES

Let $G = (V, E)$ be a simple graph with $|V| = n$ vertices and $|E| = m$ edges. A vertex v is said to be a neighbor of (or adjacent to) vertex u if there is an edge between u and v, i.e., $(u, v) \in E$. Denote by $N_G(u) = \{v \in V \mid (u, v) \in E\}$ and $d_G(u) = |N_G(u)|$ the neighbor set and the degree of the vertex u in G, respectively. Given a vertex subset $S \subseteq V$, we use $G[S]$ to denote the subgraph induced by S, i.e., $G[S] = (S, \{(u, v) \in E \mid u, v \in S\})$, and use $N_G(u, S)$ (resp. $\overline{N}_G(u, S)$) to denote sets of neighbors (resp. non-neighbors that include u itself) of u in $G[S]$. We omit the subscript G when the context is clear. Given a graph g, we use $V(g)$ and $E(g)$ to denote the sets of vertices and edges in g, respectively.

In this paper, we focus on the cohesive subgraph of *k-plex*.

Definition 2.1 (k-plex [48]). Given a positive integer k, a graph g is said to be a *k-plex* if $d_g(u) \geq |V(g)| - k$ for each vertex $u \in V(g)$.

Obviously, a 1-plex is a clique where each two vertices are adjacent. Note also that k-plex has the *hereditary* property, i.e., any induced subgraph of a k-plex is also a k-plex [48].

Problem statement. Given a graph $G = (V, E)$ and an integer $k \geq 2$, the *maximum k-plex search problem* aims to find the largest k-plex $G[S]$ with $|S| \geq 2k - 1$ in G.

Following the previous studies [11, 53], we focus on finding k-plexes with at least $2k - 1$ vertices for the following considerations. First, the value of k is usually small in real applications, e.g., $k \leq 6$ in [27, 36, 56, 63]. Hence, a k-plex with at most $2k - 2$ vertices is less informative in practice. Second, a k-plex with at least $2k - 1$ vertices has the diameter of at most 2 [63], which is more cohesive.

We next introduce some useful concepts used in this paper.

k-core/k-truss. We review useful cohesive subgraph definitions.

Definition 2.2. Given a positive integer k, a graph g is said to be

- a *k-core* if $d_g(u) \geq k$ for each vertex $u \in V(g)$ [47];
- a *k-truss* if each edge $(u, v) \in E(g)$ belongs to at least $k - 2$ triangles, i.e., $|N_g(u) \cap N_g(v)| \geq k - 2$ for each edge $(u, v) \in E(g)$ [16].

Clearly, a k-core g is a $\big(|V(g)| - k\big)$-plex and a k-truss g' is a $\big(|V(g')| - k + 1\big)$-plex.

Degeneracy order. The sequence of vertices $v_1, v_2, ..., v_n$ is called the *degeneracy order* of G if v_i has the minimum degree in subgraph $G[\{v_i, v_{i+1}, ..., v_n\}]$ for each v_i in V [4], which can be obtained in $O(m)$ [4]. Further, the *degeneracy* of G, denoted by $\delta(G)$ (or δ if the context is clear), is defined as the smallest number such that every induced subgraph of G has a vertex of degree at most $\delta(G)$.

Algorithm 1: Our framework: kPEX

Input: A graph $G = (V, E)$ and an integer k
Output: The largest k-plex $G[S^*]$
/* Stage-I.1: Heuristic&Preprocessing (Sec. 5) */
1 $S^* \leftarrow$ a large k-plex via a heuristic process KPHeuris;
2 CF-CTCP$(G, \emptyset, |S^*| - k, |S^*| - 2k, true)$;
/* Stage-I.2: Divide-and-conquer framework */
3 **while** $V(G) \neq \emptyset$ **do**
4 $\quad v \leftarrow$ the vertex with the minimum degree in G;
5 $\quad g \leftarrow$ the subgraph of G induced by $N^{\leq 2}(v)$;
$\quad$ /* Stage-II:branch-reduction-bound (Sec. 4) */
6 $\quad$ BRB_Rec$(g, \{v\}, V(g) \setminus \{v\}, k)$;
7 $\quad$ **if** *$|S^*|$ is updated due to Line 6* **then** $lb\_changed \leftarrow true$ **else** $lb\_changed \leftarrow false$;
8 $\quad$ CF-CTCP$(G, \{v\}, |S^*| - k, |S^*| - 2k, lb_changed)$;
9 **return** $G[S^*]$;
10 **Procedure** **BRB_Rec**(G, S, C, k)
11 $\quad C^\star, UB^\star \leftarrow$ AltRB(G, S, C, k);
12 $\quad$ **if** $UB^\star \leq |S^*|$ **then return**;
13 $\quad$ **if** *$S \cup C^\star$ is a k-plex* **then** update S^* by $S \cup C^\star$ and **return**;
14 $\quad v^* \leftarrow$ a branching vertex selected from $C^\star$;
15 $\quad$ BRB_Rec$(G, S \cup \{v^*\}, C^\star \setminus \{v^*\}, k)$;
16 $\quad$ BRB_Rec$(G, S, C^\star \setminus \{v^*\}, k)$;

3 THE FRAMEWORK OF KPEX

Our algorithm, named kPEX, follows the *branch-reduction-and-bound* (BRB) framework which is (conceptually) adopted by existing algorithms [11, 12, 27, 35, 36, 53, 63]. The idea is to recursively partition the current problem instance of finding the largest k-plex into two subproblem instances via a process of *branching*. Specifically, a problem instance (or branch) is denoted by (G, S, C) (or, simply (S, C) when the context is clear) where the *partial solution* S induces a k-plex (i.e., $G[S]$) and the *candidate set* C is a set of vertices that will be used to expand S. Solving the branch (S, C) refers to finding the largest k-plex $G[H]$ in the branch; *a k-plex is in the branch (S, C) if and only if $S \subseteq H \subseteq S \cup C$.* To solve a branch (S, C), it recursively solves two sub-branches formed based on a *branching vertex v* selected from C: one branch $(S \cup \{v\}, C \setminus \{v\})$ includes v to the partial solution S (which finds the largest k-plex containing v in (S, C)), and the other $(S, C \setminus \{v\})$ discards v from the candidate set C (which finds the largest k-plex excluding v in (S, C)). Clearly, solving two formed sub-branches solves branch (S, C), and solving the branch $(\emptyset, V)$ finds the largest k-plex in G.

Our kPEX adopts a similar framework in [11], which is summarized in Algorithm 1 and involves two stages. **Stage-I** first includes, in Stage-I.1, a heuristic method called KPHeuris for computing a large k-plex $G[S^*]$ (maintained globally as the largest k-plex seen so far), which will be used to narrow down the search space (Line 1), and a reduction method called CF-CTCP for reducing the input graph G by removing unpromising vertices/edges that will not appear in any k-plex larger than $|S^*|$ (Line 2). Besides, kPEX employs a widely-used *divide-and-conquer* strategy in Stage-I.2, which divides the problem of finding the largest k-plex in G into several sub-problems (Lines 3-9). Each sub-problem corresponds to a vertex v in G and aims to find the largest k-plex that (1) includes vertex v and (2) is in a subgraph of G induced by v's two-hop neighbours $N^{\leq 2}(v)$, i.e., the set of vertices that have distance at most 2 from v (note that a k-plex with at least $2k - 1$ vertices has the diameter of at most 2 [63] and thus the largest k-plex containing v is a subset of $N^{\leq 2}(v)$). We note that after solving each sub-problem, we can remove one vertex v from G since all k-plexes containing v have been found in Line 6. Clearly, the largest k-plex in G is the largest one among those returned by all sub-problems. **Stage-II** corresponds to the recursive process of solving a branch (Lines 10-16). Specifically, BRB_Rec recursively branches as discussed above (Lines 14-16). Besides, BRB_Rec conducts the newly proposed *alternated reduction-and-bound* process (AltRB) on a branch (S, C) for *narrowing down* the search space (Line 11). Specifically, it refines C to $C^\star$ by removing some unpromising vertices and computes an upper bound $UB^\star$ of (the size of) the largest k-plex in (S, C) for terminating the branch. Finally, we can terminate the branch when (1) $UB^\star \leq |S^*|$ since no larger k-plex is in the branch and (2) $S \cup C^\star$ is a k-plex since $G[S \cup C^\star]$ is the largest k-plex in the branch. We remark that a larger k-plex S^* may be identified in Line 6, which means that more unpromising vertices/edges might be removed by our pre-processing method CF-CTCP in Lines 7-8.

Novelty. Our framework differs from the state-of-the-art one [11] in the following aspects. First, in Stage-II, kPEX is based on the newly proposed AltRB for narrowing down the search space. Recall that existing methods conduct the reduction-and-bound (RB) process using a sequential method called SeqRB at Line 10 instead. We will show that AltRB performs better than SeqRB in Section 4. Specifically, it refines C to a *smaller* set $C^\star$ (i.e., $|C^\star| \leq |C|$) and obtains a *tighter* upper bound $UB^\star$ (i.e., $UB^\star \leq UB$). Second, in Stage-I.1, kPEX employs the novel KPHeuris and CF-CTCP which are more effective and efficient than existing competitors in Section 5.

4 OUR REDUCTION&BOUND METHOD: ALTRB

4.1 An Alternated Reduction-and-Bound Method

Recall that existing algorithms conduct the reduction-and-bound (RB) step using the sequential method SeqRB on a branch $B = (S, C)$ for narrowing down the search space. Specifically, SeqRB has two sequential procedures: 1) the *reduction process* refines the candidate set C to C' based on $|S^*|$ (i.e., the lower bound of the branch), i.e., removing from C those vertices that cannot appear in a k-plex larger than $|S^*|$; and 2) the *bounding process* obtains the upper bound of the largest k-plex in the refined branch (S, C'), i.e., the upper bound of the branch denoted by $UB(S, C')$. In this paper, we propose a new alternated reduction-and-bound method, called AltRB, which is based on a binary partition of a branch $B = (S, C)$ as below.

$$S = S_L \cup S_R, \quad C = C_L \cup C_R. \tag{1}$$

Let $G[H]$ be a k-plex in the branch B such that $G[H]$ is larger than the largest k-plex $G[S^*]$ seen so far, i.e., $|H| \geq |S^*| + 1$ (note that other k-plexes have the size at most $|S^*|$ and thus can be ignored during the exploration of the branch). Based on the above partition,

Algorithm 2: Alternated reduction-and-bound: `AltRB`

Input: A graph $G = (V, E)$, a branch (S, C) and an integer k
Output: Refined candidate set $C^\star$ and upper bound $UB^\star$

```
1 S_L, S_R, C_L, C_R ← Partition(G, S, C, k);
2 UB_L ← |C_L|, LB_L ← 0;
3 while UB_L is not equal to ComputeUB(S_L, C_L) do
4     UB_L ← ComputeUB(S_L, C_L);
5     LB_R ← (|S*| + 1) − |S| − UB_L; C_R ← RR1&RR2 on C_R;
6     UB_R ← ComputeUB(S_R, C_R);
7     LB_L ← (|S*| + 1) − |S| − UB_R; C_L ← RR1&RR2 on C_L;
8 return C* ← C_L ∪ C_R and UB* ← |S| + UB_L + UB_R;
```

a k-plex $G[H]$ in B can be divided into three parts as below.

$$H = S \cup (C_L \cap H) \cup (C_R \cap H). \tag{2}$$

We denote by LB_L and UB_L (resp. LB_R and UB_R) the lower and upper bounds of the size of $C_L \cap H$ (resp. $C_R \cap H$), respectively. Formally, we have

$$|C_L \cap H| \le UB_L,\ |C_R \cap H| \le UB_R. \tag{3}$$

Besides, we have the following lemma on the above partition.

Lemma 4.1. *Given a branch* (S, C) *with a partition, we have*

$$|C_L \cap H| \ge (|S^*|+1) - |S| - UB_R,\ |C_R \cap H| \ge (|S^*|+1) - |S| - UB_L. \tag{4}$$

Proof. This can be easily verified since otherwise if $|C_L \cap H| < (|S^*| + 1) - |S| - UB_R$, we have $|H| = |S| + |C_L \cap H| + |C_R \cap H| < |S| + (|S^*| + 1) - |S| - UB_R + UB_R = |S^*| + 1$, which contradicts with $|H| \ge |S^*| + 1$. A similar contradiction can be derived for the other case $|C_R \cap H| < (|S^*| + 1) - |S| - UB_L$. □

Based on Lemma 4.1, we define LB_L and LB_R as follows.

$$(|S^*| + 1) - |S| - UB_R \le LB_L \le |C_L \cap H| \tag{5}$$

$$(|S^*| + 1) - |S| - UB_L \le LB_R \le |C_R \cap H| \tag{6}$$

We note that Lemma 4.1 and Equations (5) and (6) indicate the relation between the lower bound of one part and the upper bound of the other, which enables `AltRB`. We summarize `AltRB` in Algorithm 2, which *iteratively and alternatively* conducts the reduction-and-bound step on the two partitions obtained via `Partition` (Line 1). Specifically, after initializing UB_L and LB_L in Line 2, `AltRB` involves the following steps (the details of the two procedures `Partition` and `ComputeUB` are provided in Section 4.2).

- **Step 1 (Bound on C_L).** Compute the upper bound for C_L (i.e., UB_L) via a procedure `ComputeUB` (Line 4).
- **Step 2 (Reduction on C_R).** Update the lower bound for C_R (i.e., LB_R) by $(|S^*|+1) - |S| - UB_L$ according to **Lemma 4.1** and then refine C_R based on the updated bounds via reduction rules **RR1** and **RR2** (Line 5).
- **Step 3 (Bound on C_R).** Compute the upper bound for the refined C_R (i.e., UB_R) via a procedure `ComputeUB` (Line 6).
- **Step 4 (Reduction on C_L).** Update the lower bound for C_L (i.e., LB_L) by $(|S^*|+1) - |S| - UB_R$ according to **Lemma 4.1** and then refine C_L based on the updated bounds via reduction rules **RR1** and **RR2** (Line 7).

Finally, we repeat Steps 1-4 until UB_L remains unchanged (Line 3). We remark that once tighter upper bounds are obtained at Step 1 and Step 3, tighter lower bounds can be derived via Lemma 4.1 at Step 2 and Step 4 which will be used to boost the performance of **RR1** and **RR2**. Below find the details of reduction rules.

RR1. Given a branch (S, C) with LB_L and LB_R, 1) for a vertex v in C_L, we remove v from C if $|N(v, S \cup C_L)| < LB_L + |S| - k$ or $|N(v, S \cup C_R)| < LB_R + |S| - k + 1$; and 2) for a vertex v in C_R, we remove v from C if $|N(v, S \cup C_L)| < LB_L + |S| - k + 1$ or $|N(v, S \cup C_R)| < LB_R + |S| - k$.

RR2. Given a branch (S, C) with UB_L and UB_R, 1) if $UB_L + UB_R + |S| = |S^*| + 1$ and $UB_L = |C_L|$, we move all vertices in C_L from C to S if $G[S \cup C_L]$ is a k-plex; otherwise, i.e., it is not a k-plex, we terminate the branch (S, C); 2) if $UB_L + UB_R + |S| = |S^*| + 1$ and $UB_R = |C_R|$, we move all vertices in C_R from C to S if $G[S \cup C_R]$ is a k-plex; otherwise, i.e., it is not a k-plex, we terminate the branch (S, C).

Benefits. Before proving the correctness, we show that `AltRB` better narrows down the search space than the existing `SeqRB`. The rationale behind is based on the following observations. First, at **Step 2** and **Step 4**, **RR1**, and **RR2** (which are based on UB_L, UB_R, LB_L and LB_R) will remove from C more vertices when the lower bounds LB_L and LB_R become larger and/or the upper bounds UB_L and UB_R become smaller; Second, at **Step 1** and **Step 3**, with some vertices being removed from C_L and C_R, smaller upper bound UB_L and UB_R can be derived via `ComupteUB` (details refer to Section 4.2), and larger lower bounds LB_L and LB_R can also be obtained via Lemma 4.1; Third, as `AltRB` iteratively proceeds, the bounding process and the reduction process will benefit each other (since the former will derive smaller upper bounds and larger lower bounds after the latter while the latter will remove more vertices from C after the former). In contrast, `SeqRB` cannot be conducted iteratively since (1) its reduction rules are only based on $|S^*|$, which will not be changed after `SeqRB` and (2) thus repeating it multiple times cannot result in either a smaller candidate set C or a smaller upper bound. We remark that the refined set $C^\star$ and the upper bound $UB^\star$ obtained by `AltRB` is potentially *smaller* than those obtained by `SeqRB` (which will be proved in Section 4.2). Thus, with the proposed `AltRB`, our algorithm `kPEX` runs up to *two orders of magnitude faster* than the state-of-the-arts, as verified in our experiments.

Correctness. We then show the correctness of `AltRB`. Note that `AltRB` admits an arbitrary partition on (S, C) and any possible procedure for computing UB_L and UB_R that satisfy Equation (3).

The correctness of **RR1** can be proved by contradiction. Consider a k-plex $G[H]$ in branch B with $|H| \ge |S^*| + 1$. Note that if such a k-plex does not exist, **RR1** is obviously correct since all k-plexes in branch B are no larger than $|S^*|$ and thus branch B can be terminated. In general, there are two cases. First, assume that $G[H]$ contains a vertex v in C_L such that $|N(v, S \cup C_L)| < LB_L + |S| - k$. We get the contradiction by showing that v has more than k non-neighbours in H and thus $G[H]$ is not a k-plex since $|N(v, H)| = |N(v, H \cap (S \cup C_L))| + |N(v, H \cap C_R)| \le (LB_L + |S| - k - 1) + |H \cap C_R| \le (|S| + |H \cap C_R| + |H \cap C_L|) - (k+1) = |H| - (k+1)$. Second, assume that $G[H]$ contains a vertex v in C_L such that $|N(v, S \cup C_R)| < LB_R + |S| - k + 1$. Similarly, we derive the contradiction by showing that v has more than k non-neighbours in H and thus $G[H]$ is not a k-plex since $|N(v,H)| = |N(v, H\cap(S\cup C_R))| + |N(v, H\cap C_L)| \le (LB_R+|S|-k)+(|H\cap C_L|-1) \le (|S|+|H\cap C_R|+|H\cap C_L|)-(k+1) = |H|-(k+1)$ (note that $|N(v, H\cap C_L)| \le |H\cap C_L|-1$ since v is in C_L and is not adjacent to itself). Symmetrically, we can prove the correctness for the reduction rules on C_R.

The correctness of **RR2** is easy to verify. Consider a branch (S,C) with $UB_L+UB_R+|S| = |S^*|+1$ and $UB_L = |C_L|$, and a k-plex $G[H]$ in (S,C) with $|H| \ge |S^*|+1$ (note that if such a k-plex does not exist, **RR2** is obviously correct on this branch). We note that $G[H]$ must contain all vertices in C_L, i.e., $C_L \subseteq H$, since otherwise $|H| = |H\cap S|+|H\cap C_L|+|H\cap C_R| \le |S|+(|C_L|-1)+|H\cap C_R| \le |S|+UB_L+UB_R-1 = |S^*|$. Therefore, $G[S\cup C_L]$ must be a k-plex due to the hereditary property; otherwise, such a k-plex cannot exist in (S,C) and we can terminate the branch.

The correctness of AltRB can then be easily verified.

4.2 Upper Bound Computation and Greedy Partition Strategy

In this part, we first introduce the method ComputeUB used at **Step 1** and **Step 3** for obtaining UB_L and UB_R in Section 4.1. To boost the performance of ComputeUB as well as the reduction rules on C_L and C_R, we then propose a greedy strategy Partition for partitioning C (resp. S) into C_L and C_R (resp. S_L and S_R). Finally, with all carefully-designed techniques above, we show that the resulted upper bound $UB^\star$ will be potentially *smaller* than the existing one UB.

Upper bound computation. We adapt an existing upper bound computation [35], which we call ComputeUB, for obtaining UB_L and UB_R. Note that it can handle an arbitrary partition on a branch (S,C). Consider **Step 1** for computing UB_L. ComputeUB(S_L,C_L) first iteratively partitions C_L into $(|S_L|+1)$ disjoint subsets. The i-th $(1\le i\le |S_L|)$ subset $\Pi_i(S_L,C_L)$ contains all non-neighbours of a vertex $u_i \in S_L$ in $C_L - \{\Pi_1(S_L,C_L),...,\Pi_{i-1}(S_L,C_L)\}$, formally,

$$\Pi_i(S_L,C_L) = \overline{N}(u_i, C_L^i),\ C_L^i = C_L - \cup_{j=1}^{i-1}\Pi_j(S_L,C_L), \quad (7)$$

where u_i is the vertex in $S_L \setminus \{u_1,u_2,...,u_{i-1}\}$ with the largest ratio of $|\overline{N}(u_i,C_L^i)|/(k-|\overline{N}(u_i,S)|)$. Note that the strategy of selecting u_i from S_L has been shown to boost the practical performance of ComputeUB (details refer to [35]). Besides, we have $\Pi_0(S_L,C_L) = C_L - \{\Pi_1(S_L,C_L),...,\Pi_{|S_L|}(S_L,C_L)\}$. Thus, vertices in $\Pi_i(S_L,C_L)$ $(1\le i\le |S_L|)$ are the non-neighbours of u_i in C_L, and vertices in $\Pi_0(S_L,C_L)$ are common neighbours of vertices in S_L. The key observation is that for a k-plex $G[H]$ in the branch, $C_L\cap H$ *contains at most* $\min\{|\Pi_i(S_L,C_L)|, k-|\overline{N}(u_i,S)|\}$ *vertices from* $\Pi_i(S_L,C_L)$ *for* $1\le i\le |S_L|$ since otherwise u_i (in H) will have more than k non-neighbours in $G[H]$ and thus $G[H]$ is not a k-plex. Thus, the upper bound UB_L returned by ComputeUB(S_L,C_L) gives as below:

$$|\Pi_0(S_L,C_L)| + \sum_{i=1}^{|S_L|}\min\{|\Pi_i(S_L,C_L)|, k-|\overline{N}(u_i,S)|\}. \quad (8)$$

We note that with some vertices being removed from C_L during AltRB, $\Pi_i(S_L,C_L)$ will get smaller and thus a smaller upper bound can be derived. Similarly, we can obtain UB_R by ComputeUB(S_R,C_R). Besides, we remark that the state-of-the-art upper bound of k-plex in the branch (S,C) (used in SeqRB) is $|S|$+ComputeUB(S,C) [35].

Greedy partition. We first note that finding the "best" partition towards pruning the largest number of vertices from the candidate set C is hard to solve since (1) the newly proposed reduction rules **RR1** and **RR2** are non-trivial and depend on the largest k-plex S^* seen so far, and (2) the number of possible binary partitions is exponentially large, i.e., $O(2^{|S|+|C|})$. As a result, it is theoretically untraceable how different partition methods affect the overall performance of kPEX. Hence, we propose to explore heuristic partitions towards achieving better practical performance.

Consider the upper bound computation at C_L (Equation (8)). We observe that *all vertices in* $\Pi_0(S_L,C_L)$ *contribute to the upper bound* ComputeUB(S_L,C_L) since each of them connects to all vertices in S_L; thus, they could appear in a k-plex in branch (S,C). The similar observation can be derived on other subsets $\Pi_i(S_L,C_L)$ such that $|\overline{N}(u_i,C_L^i)| \le k-|\overline{N}(u_i,S)|$ and $1\le i\le |S_L|$ (note that there are *fewer* missing edges between S_L and those subsets). Thus, the adapted upper bound computation performs worse on those subsets.

Motivated by the above observation, we propose to divide S and C into the one (S_L and C_L) with *more* missing edges and the other (S_R and C_R) with *fewer* missing edges. We summarize the proposed strategy in Algorithm 3. Specifically, we iteratively remove from S to S_L (resp. from C to C_L) the vertex v with the greatest value of $|\overline{N}(v,C)|/(k-|\overline{N}(v,S)|)$ (resp. the set of v's non-neighbours in C, i.e., $\overline{N}(v,C)$) until the greatest value of $|\overline{N}(v,C)|/(k-|\overline{N}(v,S)|)$ is not greater than 1 or S becomes empty (Lines 2-6). Then, all remaining vertices in S and C will be removed to S_R and C_R (Line 7). We observe that (1) ComputeUB(S_L,C_L) will return a tighter bound since $|\overline{N}(u_i,C_L^i)| > k-|\overline{N}(u_i,S)|$ holds for $1\le i\le |S_L|$ and $\Pi_0(S_L,C_L)=\emptyset$, and (2) ComputeUB(S_R,C_R) is always equal to $|C_R|$.

Consider a branch (S,C) (which has been refined by SeqRB) with the upper bound $UB = |S|$+ComputeUB(S,C). With the proposed techniques, AltRB will further narrow down the search space of (S,C) by the following observation.

$$UB^\star \le UB \text{ and } |C^\star| \le |C|. \quad (9)$$

We note that $|C^\star| \le |C|$ is obvious since some vertices in C could be removed via **RR1** and **RR2**. Besides, $UB^\star \le UB$ holds since (1) ComputeUB(S,C) =ComputeUB(S_L,C_L)+ComputeUB(S_R,C_R) before AltRB (which can be verified based on the definitions) and (2) as AltRB proceeds, C_L and C_R are refined via **RR1** and **RR2**, and thus ComputeUB(S_L,C_L) and ComputeUB(S_R,C_R) get smaller.

Benefits of greedy partition. Compared with a random partition, the greedy partition in Algorithm 3 has the following advantageous properties. (1) A tight upper bound of C_L leads to a larger LB_R, which enhances the effectiveness of **RR1**. (2) $UB_R = |C_R|$ is always satisfied, which means that **RR2** is applicable as long as UB_L+UB_R+$|S| = |S^*|+1$. In other words, the conditions for **RR2** are more relaxed. Moreover, computing UB_R as $|C_R|$ is easy to implement and requires less computation. In addition, we remark that with our proposed greedy partition method Partition, AltRB can prune more vertices from C compared to SeqRB in theory, as shown in Equation (9), while other partition methods may not.

4.3 Summary and Analysis

Time complexity. We analyze the time complexity of AltRB as follows. (1) AltRB first invokes Partition (Algorithm 3). Specifically, Lines 2-6 of Algorithm 3 will be conducted at most $|S|$ times, and each iteration needs to compute $|\overline{N}(v,C)|$ for each $v \in S$,

Algorithm 3: `Partition`(G, S, C, k)

Input: Branch (S, C), a graph $G = (V, E)$, and an integer k
Output: The greedy partition S_L, S_R, C_L and C_R

```
S_L ← ∅, S_R ← ∅, C_L ← ∅, C_R ← ∅;
while S ≠ ∅ do
    v* ← arg max_{v∈S} |N̄(v, C)|/(k − |N̄(v, S)|) ;
    if |N̄(v*, C)|/(k − |N̄(v*, S)|) ≤ 1 then break;
    S_L ← S_L ∪ {v*}, C_L ← C_L ∪ N̄(v*, C);
    S ← S \ {v*}, C ← C \ N̄(v*, C);
S_R ← S, C_R ← C;
return S_L, S_R, C_L and C_R;
```

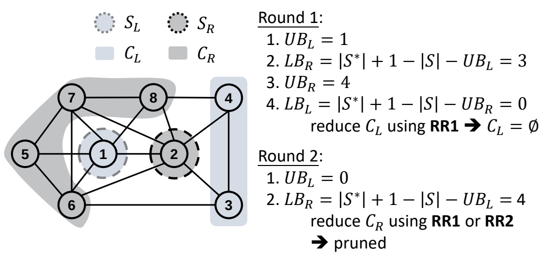


Figure 1: An example of `AltRB` with $k = 2$, $|S^*| = 5$, $S = \{v_1, v_2\}$, $C = \{v_3, v_4, v_5, v_6, v_7, v_8\}$

which can be done in $O(|S| \times |C|)$. Thus, the time complexity of `Partition` is $O(|S|^2|C|)$. (2) `AltRB` then iteratively processes Lines 3-7 of Algorithm 2. We note that `ComputeUB`(S, C) can be computed in $O(|S|^2|C|)$ [35] (Lines 4 and 6). For reductions rules in Lines 5 and 7, **RR1** iteratively removes the vertex in C_R with minimum $|N(v, S \cup C_L)|$ (or $|N(v, S \cup C_R)|$), and **RR2** checks whether $S \cup C_R$ is a k-plex. Both rules can be done in $O(|C| \times (|S| + |C|))$. (3) We also know that $|S|$ is bounded by $\delta(G) + k$; otherwise, we have a k-plex $G[S]$ with $|S| > \delta(G) + k$, which will form a $(\delta(G) + 1)$-core and thus contradict the definition of $\delta(G)$; $|C|$ is bounded by $\delta(G)d$ since $V(g)$ at Line 5 of Algorithm 1 is bounded by $\delta(G)d$ [17, 54]. (4) Let r be the number of iterations of Lines 3-7 of Algorithm 2. The number of r is quite small in practice (e.g., $r = 1.13$ on average in our experiments) and is bounded by $|C|$ (i.e., $\delta(G)d$) since at least one vertex is removed from C in each round until C becomes empty.

Thus, `AltRB` (Algorithm 2) runs in $O(r \times (|S|^2|C| + |C|^2)) = O(\delta(G)^3d^3 + k^2\delta(G)^2d^2)$, where $\delta(G)$ is much smaller than d and n in real graphs, as shown in Table 1 ($\delta(G) \le d < n$ in theory).

Space complexity. We note that our `kPEX` has the space complexity of $O(m + n)$, which is the same as the SOTA algorithm `kPlexT`. In specific, in Stage-I, `CF-CTCP` needs to maintain two sets Q_v and Q_e, which have the size of $O(n + m)$ and thus dominate the space complexity. In Stage-II, `BRB_Rec` recursively maintains three global data structure, namely g, S, and C, for each branch, which dominate the space complexity. Here, graph g is obtained at Line 5 of Algorithm 1 and thus has the size bounded by $O(n + m)$. S and C are two disjoint vertex sets and have the size bounded by $O(n)$ clearly.

Example. To illustrate the proposed `AltRB`, consider an example in Figure 1 with k=2, $|S^*| = 5$, $S = \{v_1, v_2\}$ and $C = \{v_3, v_4, v_5, v_6, v_7, v_8\}$. First, we apply the greedy partition (Algorithm 3) and obtain $S_L = \{v_1\}$, $S_R = \{v_2\}$, $C_L = \{v_3, v_4\}$, and $C_R = \{v_5, v_6, v_7, v_8\}$. Then, in the first round of `AltRB` (Lines 4-7 in Algorithm 2), we conduct the four steps. (**Step 1**) Compute the upper bound of C_L, i.e., $UB_L = 1$. (**Step 2**) Update the lower bound of C_R (i.e., $LB_R = 3$) and reduce C_R via **RR1** and **RR2** (no vertices are removed). (**Step 3**) Compute the upper bound of C_R, i.e., $UB_R = 4$. (**Step 4**) Update the lower bound of C_L (i.e., $LB_R = 0$) and reduce C_L via **RR1** and **RR2** (C_L is reduced to an empty set). Next, in the second round with $C_L = \emptyset$, we (1) compute $UB_L = 0$, and (2) update $LB_R = 4$ and reduce C_R. If we first apply **RR1** to C_R, v_5, v_6, v_7 and v_8 will be removed, and finally compute the upper bound as $UB^\star = |S| + |UB_L| + |UB_R| = 2 + 0 + 0 = 2$, resulting in pruning. If we first apply **RR2**, both $UB_L + UB_R + |S| = |S^*| + 1$ and $UB_R = |C_R|$ are satisfied. We then find that $G[S \cup C_R]$ is not a k-plex, which means that **RR2** also leads to pruning. Actually, the size of maximum 2-plex is 5, indicating that the branch (S, C) cannot find a larger 2-plex, and thus this branch can be pruned by `AltRB`. However, without `AltRB`, the existing method [35] will compute an upper bound as $UB = UB_L + UB_R + |S| = 1 + 4 + 2 = 7$, which cannot prune the current branch.

Remarks. We remark that the existing reduction rules proposed in [11, 12, 53] are all based on $|S^*|$ and thus orthogonal to `AltRB`. We conduct some of these reduction rules to improve practical performance, including (1) additional reduction on subgraph g (Lemma 3.2 in [11] and Reduction 2 in [53]) in Line 5 of Algorithm 1, and (2) reduction on C before `AltRB` (**RR4** in [11] and Algorithm 3 in [12]). Besides, `AltRB` is also orthogonal to the branching rules for selecting the branching vertex and forming the sub-branches.

5 EFFICIENT PRE-PROCESSING TECHNIQUES

In this section, we develop some efficient pre-processing techniques for further boosting the performance of BRB algorithms, namely, `CF-CTCP` for reducing the size of the input graph in Section 5.1 and `KPHeuris` for heuristically computing a large k-plex in Section 5.2. We note that `CF-CTCP` and `KPHeuris` are *orthogonal* to the branch-and-bound methods including our `kPEX`. In specific, as shown in Algorithm 1, they are conducted at Stage-I, which are prior to applying the branch-and-bound method `kPEX` at Stage-II. Besides, we note that with `KPHeuris` and/or `CF-CTCP`, our `kPEX` would achieve better performance, as verified in the experiments, i.e., Table 5.

5.1 Faster Core-Truss Co-Pruning: `CF-CTCP`

Let lb be the lower bound of the size of the largest k-plex (which corresponds to the size of the largest k-plex $G[S^*]$ seen so far). We also let $\Delta(u, v)$ be the set of common neighbors of u and v in G, i.e., $\Delta(u, v) = N_G(u) \cap N_G(v)$. The idea of refining the input graph G is to *remove from G those vertices and edges that cannot appear in any k-plex larger than lb as many as we can.* Existing methods [11, 35, 63] are all based on the following lemmas and differ in the implementations (the details of proof is omitted).

Lemma 5.1. *(Core Pruning [27]) For each vertex $u \in V(G)$, u cannot appear in a k-plex of size $lb + 1$ if $d_G(u) \le lb - k$.*

Lemma 5.2. *(Truss Pruning [63]) For each edge $(u, v) \in E(G)$, (u, v) cannot appear in a k-plex of size $lb + 1$ if $\delta_G(u, v) \le lb - 2k$ where $\delta_G(u, v)$ is the number of common neighbors of u and v, i.e., $\delta_G(u, v) = |\Delta(u, v)|$.*

Note that the time complexities of core pruning and truss pruning are $O(m)$ [4] and $O(m \times \delta(G))$ [51], respectively. The above two lemmas identify those unpromising vertices and edges that can be removed from G. In particular, with some vertices or edges being removed from G, the remaining vertices u and edges (u, v) have $d_G(u)$ and $\delta_G(u, v)$ decreases, respectively; and then more vertices and edges can be removed. Therefore, the core pruning (resp. the truss pruning) can be conducted in an iterative way.We remark that the state-of-the-art method called the core-truss co-pruning (CTCP [11]) iteratively conducts the truss pruning and then the core pruning in multiple rounds until the graph remains unchanged. However, we observe that CTCP is still inefficient due to potential redundant computations. This is because (1) CTCP performs the truss pruning and the core pruning *separately* at each round (i.e., first remove a set of edges via the truss pruning and then remove one unpromising vertex via core pruning), (2) the truss pruning has the time complexity of $O(m \times \delta(G))$ lager than $O(m)$ for the core pruning, and (3) we note that during the truss pruning, some vertices can be removed via the more efficient core pruning while the truss pruning will iteratively check all their incident edges and then remove some of them (which is very costly).

To improve the practical efficiency of CTCP, we propose a new algorithm called the *core-pruning-first core-truss co-pruning* (or CF-CTCP), which differs from CTCP in the way of conducting pruning at each round. Specifically, at each round, it first removes *all* unpromising vertices and then removes *one* unpromising edge (recall that CTCP first removes *all* unpromising edges and then *one* unpromising vertex). The benefit is that unpromising vertices can be removed immediately via efficient core pruning. Note that our CF-CTCP has the same output but requires less computation compared to CTCP. Given the integer k and the lower bound lb, both CF-CTCP and CTCP reduce the input graph G to the maximal subgraph that is $(lb + 1 - k)$-core and $(lb + 3 - 2k)$-truss.

The main idea of CF-CTCP is to conduct core pruning thoroughly as follows: 1) if we identify an edge that can be removed, we will immediately remove this edge, even if we have not yet finished computing $\Delta(\cdot, \cdot)$ (i.e., all triangles for each edge); 2) after removing an edge (u, v), we will check whether u or v can be reduced by core pruning. Note that after removing an edge (u, v), we postpone the action of updating $\Delta(u, \cdot)$ and $\Delta(v, \cdot)$ since it is time-consuming and there may lead to redundant computations. For example, if both vertices u and v will be removed by core pruning later, updating $\Delta(u, \cdot)$ and $\Delta(v, \cdot)$ is not necessary.

Algorithm 4: CF-CTCP($G = (V, E), Q_v, \tau_v, \tau_e, lb\_changed$)

Input: A graph $G = (V, E)$, the set of vertices to be removed Q_v, two integral thresholds τ_v and τ_e, a boolean value *lb_changed*

Output: The reduced graph which is the maximal subgraph in G that is both a $(\tau_v + 1)$-core and a $(\tau_e + 3)$-truss

1 Remove the vertices in Q_v from G and reduce G to the maximal $(\tau_v + 1)$-core by core pruning;
2 Initialize the set of removed edges to be considered $Q_e \leftarrow \{$*edges removed at Line 1*$\}$;
3 **if** *lb_changed* **then**
4   **for each** $(u, v) \in E$ **do**
5   **if** CF-CTCP *is invoked for the first time* **then**
6             $\Delta(u, v) \leftarrow N_G(u) \cap N_G(v)$;
7   **if** $|\Delta(u, v)| \leq \tau_e$ **then**
8             $Q_e \leftarrow Q_e \cup$ RemoveEdge($G, (u, v), \tau_v$);
9 **while** $Q_e \neq \emptyset$ **do**
10     $(u, v) \leftarrow$ pop an edge from Q_e;
11   **if** $u \in V$ **then**
12   **for each** $w \in N_G(u)$ *satisfying* $v \in \Delta(u, w)$ **do**
13   Remove v from $\Delta(u, w)$;
14   **if** $|\Delta(u, w)| \leq \tau_e$ **then**
15                 $Q_e \leftarrow Q_e \cup$ RemoveEdge($G, (u, w), \tau_v$);
16   **if** $v \in V$ **then**
17   **for each** $w \in N_G(v)$ *satisfying* $u \in \Delta(v, w)$ **do**
18   Remove u from $\Delta(v, w)$;
19   **if** $|\Delta(v, w)| \leq \tau_e$ **then**
20                 $Q_e \leftarrow Q_e \cup$ RemoveEdge($G, (v, w), \tau_v$);

Procedure: RemoveEdge($G, (u, v), \tau_v$)

Output: The set of removed edges to be considered Q_e

21 Remove the *unpromising edge* (u, v) from G;
22 Reduce G to the maximal $(\tau_v + 1)$-core by core pruning;
23 Initialize the set of removed edges to be considered $Q_e \leftarrow \{$*edges removed at Lines 21-22*$\}$;
24 **return** Q_e;

Our proposed CF-CTCP is shown in Algorithm 4. The input of CF-CTCP includes: 1) a set of vertices Q_v, which stores the vertices that need to be removed; 2) two integers $\tau_v = lb - k$ and $\tau_e = lb - 2k$ that serve as thresholds for the numbers of degrees and triangles for pruning, respectively; 3) a boolean value *lb_changed* which is *true* if a larger k-plex is found in kPEX and KPHeuris (Algorithm 1 and Algorithm 5). We note that both kPEX and KPHeuris (Algorithms 1 and 5) invoke CF-CTCP multiple times. For example, KPHeuris invokes CF-CTCP by calling CF-CTCP($G, \emptyset, lb - k, lb - 2k, true$) when it finds a larger heuristic k-plex of size lb.

We then describe the details of CF-CTCP in steps. First, we design a procedure called RemoveEdge (Lines 21-24) to remove one unpromising edge in Line 21 and all current unpromising vertices in Line 22 via core and truss pruning. The set of removed edges to be considered (due to Lines 21 and 22) is pushed into Q_e, which will be used to update $\Delta(\cdot, \cdot)$ later. Second, Lines 5-6 initialize the sets of common neighbours $\Delta(\cdot, \cdot)$ if CF-CTCP is invoked for the first time. Whenever we find an edge (u, v) that can be reduced, we invoke the procedure RemoveEdge to remove (u, v) immediately in Line 8. Third, we postpone the action of updating $\Delta(\cdot, \cdot)$ to Lines 9-20. Lines 11-20 consider the effect of each removed edge (u, v) by traversing all the triangles that (u, v) participates in. Specifically, Lines 11-15 traverse each edge $(u, w) \in E$ satisfying $v \in \Delta(u, w)$, i.e., u, v, w can form a triangle, then we update $\Delta(u, w)$ and check whether edge (u, w) can be reduced. Lines 16-20 consider the edges connected to v, which is similar to Lines 11-15. Note that in Lines 15 and 20, if we find an edge that can be reduced, we invoke the procedure RemoveEdge to remove the edge immediately.

Algorithm 5: KPHeuris(G, k)

Input: A graph $G = (V, E)$ and an integer $k > 1$

Output: The vertex set S of a heuristic initial k-plex $G[S]$

1 $S \leftarrow$ Degen(G, k), $lb \leftarrow |S|$;
2 Apply CF-CTCP for refining G based on lb;
3 **for each** $v_i \in V(G)$ **do**
4     $g \leftarrow G[\{v_i, v_{i+1}, ..., v_n\} \cap N^{\leq 2}(v_i)]$; $S' \leftarrow$ Degen(g, k);
5   **if** $|S'| > |S|$ **then**
6         $S \leftarrow S'$, $lb \leftarrow |S|$;
7   Apply CF-CTCP for refining G based on lb;
8 **return** S;

Procedure: Degen(G, k)

Output: The vertex set S of a heuristic maximal k-plex in G

9 $v_1, v_2, ..., v_n \leftarrow$ the degeneracy order of vertices in $V(G)$;
10 $S \leftarrow \emptyset$;
11 **for** $i = n$ *to* 1 **do**
12   **if** $G[S \cup \{v_i\}]$ *is a k-plex* **then** $S \leftarrow S \cup \{v_i\}$;
13 **return** S;

Time complexity. We analyze the time complexity of CF-CTCP (Algorithm 4), including all invocations in kPEX, in the following.

Lemma 5.3. *The total time complexity of all invocations in* kPEX *(Algorithm 1 which includes invocations in the heuristic process* KPHeuris *in Algorithm 5)) to* CF-CTCP *(Algorithm 4) is $O(m \times \delta(G))$.*

The omitted proof, along with an implementation of CF-CTCP with $O(m)$ memory usage, is provided in the technical report [29].

Remarks. First, the time complexity of CTCP is $O(m \times \delta(G) + m \times k)$=$O(m\delta(G))$, requiring that k is a small constant. However, k is up to n in theory and the time complexity of our CF-CTCP is always $O(m\delta(G))$ for all possible values of k. **Second**, we do not consider the update of $\Delta(\cdot, \cdot)$ when removing a vertex because removing a vertex is equivalent to first removing all the edges connected to this vertex and then removing this isolated vertex. Therefore, we only consider the removed edges for updating $\Delta(\cdot, \cdot)$. **Third**, the acceleration of CF-CTCP can be attributed to two main factors: 1) we do not need to compute the numbers of triangles for the edges that can be removed by core pruning; 2) for an edge (u, v) to be removed such that both u and v are already removed by core pruning, we do not need to traverse related triangles to update $\Delta(\cdot, \cdot)$. Note that if we cannot remove any vertex or edge, the time consumption of CF-CTCP will be the same as CTCP in theory, which is due to the fact that both of them need to compute $\Delta(\cdot, \cdot)$ in $O(m \times \delta(G))$.

5.2 Compute a large k-plex: KPHeuris

We introduce a heuristic method KPHeuris for computing a large initial k-plex. Note that such an initial k-plex offers a lower bound lb, which helps to narrow the search space; and the larger the lower bound is, the more search space can be refined. Therefore, KPHeuris is designed for obtaining a large k-plex *efficiently and effectively.*

We summarize KPHeuris in Algorithm 5, which relies on a subprocedure (called Degen) for computing a large k-plex. Specifically, Degen iteratively includes to an empty set S a vertex in a graph G based on the degeneracy ordering while retaining the k-plex property of $G[S]$ until we cannot continue (Lines 9-12). To compute a larger k-plexes, KPHeuris further generate n subgraphs from G, each of which corresponds to a vertex in G (Lines 3-4); it then invokes Degen on each of them to obtain a k-plex (Line 4); it finally returns the largest one among $n + 1$ found k-plexes. Note that the subgraph related to v_i is the subgraph induced by $\{v_i, v_{i+1}, ..., v_n\} \cap N^{\leq 2}(v_i)$ where $N^{\leq 2}(u)$ denotes u's neighbors and u's neighbors' neighbors, and the rationale is that it can make the subgraph smaller and denser, which tends to find a larger k-plex easier. The time complexity of Degen is $O(m)$, and we will invoke it at most $n + 1$ times, thus the total time complexity of computing heuristic solutions in Algorithm 5 is $O(nm)$. We remark that the total time complexity of all invocations of CF-CTCP is $O(m\delta(G))$ because we invoke CF-CTCP in KPHeuris only when we find a larger k-plex, i.e., $lb\_changed = true$, as shown in Lemma 5.3. Thus, the time complexity of KPHeuris is $O(m\delta(G) + nm) = O(nm)$.

Compared with existing heuristic methods. There are two state-of-the-art heuristic methods: kPlex-Degen ([11]) and EGo-Degen ([12]). kPlex-Degen computes a large k-plex by iteratively removing a vertex from the input graph G based on a certain ordering until the remaining graph becomes a k-plex. KPHeuris differs from kPlex-Degen in two aspects. First, Degen computes a large k-plex by iteratively including a vertex, which is more efficient since the size of the largest k-plex is usually much smaller than the size of the input graph (especially for real-world graphs) and can always return a maximal k-plex, while kPlex-Degen cannot. Second, KPHeuris further explores n subgraphs instead of the input graph G, which tends to obtain a larger k-plex as empirically verified in our experiments. EGo-Degen extracts a subgraph g_v for each vertex v and invokes kPlex-Degen to compute a k-plex in g_v. Then, EGo-Degen selects the largest k-plex among those computed on n subgraphs as the initial heuristic k-plex. KPHeuris differs from EGo-Degen in three aspects. First, the method of subgraph extraction is different. For a vertex $v \in V(G)$, EGo-Degen extracts $g_v = G[\{v_i, v_{i+1}, ..., v_n\} \cap N_G(v_i)]$, while our KPHeuris generates a subgraph $g'_v = G[\{v_i, v_{i+1}, ..., v_n\} \cap N^{\leq 2}(v_i)]$. It is easy to verify that $g_v \subseteq g'_v$ due to $N_G(v_i) \subseteq N^{\leq 2}(v_i)$. Additionally, a larger subgraph tends to contain a larger k-plex, as verified in Table 6. Second, EGo-Degen computes k-plexes by invoking kPlex-Degen, which implies that it may find a non-maximal k-plex as mentioned above. Third, once a larger k-plex is found, KPHeuris updates lb and removes unpromising vertices/edges immediately, while EGo-Degen does not reduce the graph until n heuristic k-plexes are computed.

6 EXPERIMENTAL STUDIES

We test the efficiency and effectiveness of our algorithm kPEX by comparing with the state-of-the-art BRB algorithms, namely, **kPlexS** ([11]), **KPLEX** ([53]), **DiseMKP** ([35]) and **kPlexT** ([12]).

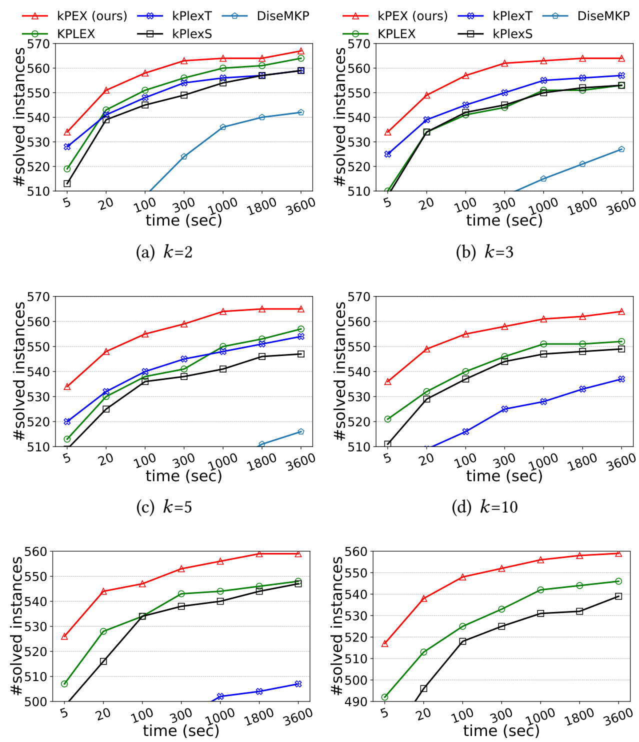
Setup. For all baselines, we use the source code from the authors. All algorithms are written in C++ and compiled with -O3 optimization by g++ 9.4.0. Moreover, all algorithms are initialized with a lower bound of $2k - 2$ to focus on finding k-plexes with at least $2k - 1$ vertices. All experiments are conducted in the single-thread mode on a machine with an Intel(R) Xeon(R) Platinum 8358P CPU@2.60GHz and 256GB main memory. The CPU frequency is

Table 1: Statistics of 30 representative graphs

ID	Graph	n	m	density	d_{max}	$\delta(G)$
G1	johnson8-4-4	70	1855	$7.68 \cdot 10^{-1}$	53	53
G2	C125-9	125	6963	$8.98 \cdot 10^{-1}$	119	102
G3	keller4	171	9435	$6.49 \cdot 10^{-1}$	124	102
G4	brock200-2	200	9876	$4.96 \cdot 10^{-1}$	114	84
G5	san200-0-9-1	200	17910	$9.00 \cdot 10^{-1}$	191	162
G6	san200-0-9-2	200	17910	$9.00 \cdot 10^{-1}$	188	169
G7	san200-0-9-3	200	17910	$9.00 \cdot 10^{-1}$	187	169
G8	p-hat300-1	300	10933	$2.44 \cdot 10^{-1}$	132	49
G9	p-hat300-2	300	21928	$4.89 \cdot 10^{-1}$	229	98
G10	p-hat500-1	500	31569	$2.53 \cdot 10^{-1}$	204	86
G11	soc-BlogCatalog-ASU	10312	333983	$6.28 \cdot 10^{-3}$	3992	114
G12	socfb-UIllinois	30795	1264421	$2.67 \cdot 10^{-3}$	4632	85
G13	soc-themarker	69413	1644843	$6.83 \cdot 10^{-4}$	8930	164
G14	soc-BlogCatalog	88784	2093195	$5.31 \cdot 10^{-4}$	9444	221
G15	soc-buzznet	101163	2763066	$5.40 \cdot 10^{-4}$	64289	153
G16	soc-LiveMocha	104103	2193083	$4.05 \cdot 10^{-4}$	2980	92
G17	soc-wiki-conflict	116836	2027871	$2.97 \cdot 10^{-4}$	20153	145
G18	soc-google-plus	211187	1141650	$5.12 \cdot 10^{-5}$	1790	135
G19	soc-FourSquare	639014	3214986	$1.57 \cdot 10^{-5}$	106218	63
G20	rec-epinions-user-ratings	755760	13667951	$4.79 \cdot 10^{-5}$	162179	151
G21	soc-wiki-Talk-dir	1298165	2288646	$2.72 \cdot 10^{-6}$	100025	119
G22	soc-pokec	1632803	22301964	$1.67 \cdot 10^{-5}$	14854	47
G23	tech-ip	2250498	21643497	$8.55 \cdot 10^{-6}$	1833161	253
G24	ia-wiki-Talk-dir	2394385	4659565	$1.63 \cdot 10^{-6}$	100029	131
G25	sx-stackoverflow	2584164	28183518	$8.44 \cdot 10^{-6}$	44065	198
G26	web-wikipedia_link_it	2790239	86754664	$2.23 \cdot 10^{-5}$	825147	894
G27	socfb-A-anon	3097165	23667394	$4.93 \cdot 10^{-6}$	4915	74
G28	soc-livejournal-user-groups	7489073	112305407	$4.00 \cdot 10^{-6}$	1053720	116
G29	soc-bitcoin	24575382	86063840	$2.85 \cdot 10^{-7}$	1083703	325
G30	soc-sinaweibo	58655849	261321033	$1.52 \cdot 10^{-7}$	278489	193

Table 2: Number of solved instances within 3600 seconds

	Network-Repository						DIMACS-2					
Solvers \ k	2	3	5	10	15	20	2	3	5	10	15	20
kPEX	**567**	**564**	**565**	**564**	**559**	**559**	**29**	**28**	**27**	**22**	**23**	**26**
KPLEX	564	553	557	552	548	546	27	23	18	17	22	21
kPlexT	559	557	554	537	507	471	25	24	17	14	20	21
kPlexS	559	553	547	549	547	539	22	20	15	15	20	21
DiseMKP	533	520	506	471	425	413	27	25	17	16	21	18



(a) k=2 (b) k=3

(c) k=5 (d) k=10

(e) k=15 (f) k=20

Figure 2: Number of solved instances on Network Repository (The lines corresponding to DiseMKP and kPlexT may not appear in the figures, as they are slow under certain settings and thus cannot reach the bottom lines within 3600 seconds.)

fixed at 3.3GHz. We set the time limit as 3600 seconds and use **OOT** (Out Of Time limit) to indicate the time exceeds the limit. We consider six different values of k, i.e., 2, 3, 5, 10, 15, and 20. We focus on the case of $k = 5$, and defer the experiments for other values of k to the appendix of the technical report [29]. We also note that the major findings for $k = 5$ hold for other values of k.

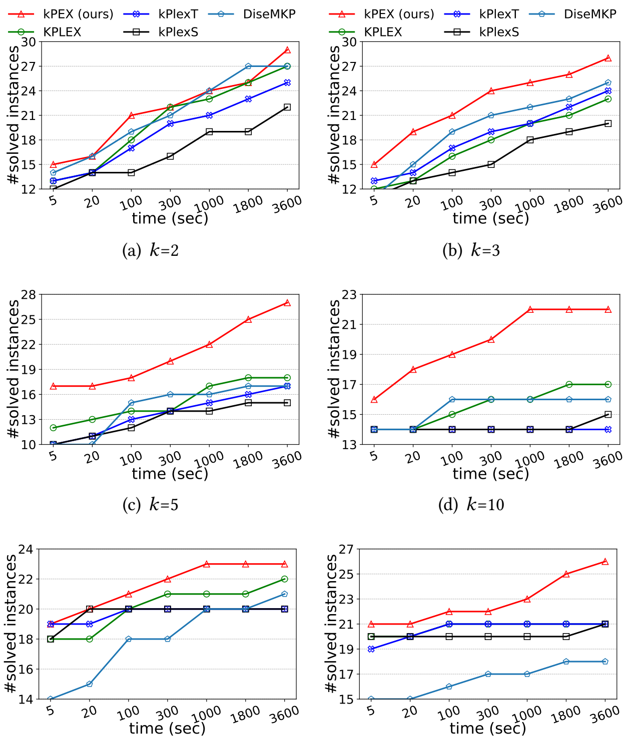
Datasets. We consider the following two collections of graphs.

- **Network Repository [2].** The dataset contains 584 graphs with up to 5.87×10^7 vertices. Most of them are real-world graphs.
- **2nd-DIMACS (DIMACS-2) Graphs [1].** The dataset contains 80 synthetic dense graphs with up to 4000 vertices. Most graphs are synthetic, which are often hard to be solved [35, 36, 53].

For better comparisons, we select 30 representative graphs from the above 664 graphs and report the statistics in Table 1, where the graph density is $\frac{2m}{n(n-1)}$ and the maximum degree is d_{max}. The criteria of selecting these representative graphs are as follows. First, following [53], we do not select extremely easy or hard graphs, i.e., those graphs that can be solved within 5 seconds by all five solvers or cannot be solved within 3600 seconds by any solver when $k = 5$. Second, the representative graphs cover a wide range of sizes. Among the selected graphs, 10 small dense graphs (G1-G10) are synthetic graphs from **DIMACS-2 Graphs**, 10 medium graphs (G11-G20) with at most 10^6 vertices, and 10 large sparse graphs (G21-G30) with at least 10^6 vertices are real-world graphs from **Network Repository**. Third, most of the representative graphs have also been selected in previous studies [36, 53, 56].

6.1 Comparing with State-of-the-art Algorithms

Number of solved instances on two collections of graphs. We compare kPEX with four baselines by reporting the numbers of solved instances. The results for **Network Repository** are shown in Table 2 and Figure 2. We observe that kPEX outperforms all baselines for all tested values of k. For example, kPEX solves 12 instances more than the best baseline KPLEX for $k = 10$ within 3600 seconds. In addition, our kPEX is more stable than baselines when varying k. In contrast, there is an obvious drop in solved instances for kPlexT and DiseMKP as k increases from 2 to 20. This demonstrates the superiority of kPEX, which employs the AltRB strategy (with novel reduction and bounding techniques) and efficient pre-processing methods. The results on the collection of **DIMACS-2 Graphs** are shown in Table 2 and Figure 3. kPEX outperforms all baselines by solving the most instances with 3600 seconds for all tested k values, e.g., kPEX solves 9 instances more than the second best solver KPLEX for $k = 5$. Besides, we note that kPEX is comparable with DiseMKP when $k = 2$. This is because the proposed reduction rules and upper bounding method are less effective for small values of k.



(a) k=2
(b) k=3
(c) k=5
(d) k=10
(e) k=15
(f) k=20

Figure 3: Number of solved instances on DIMACS-2

Running time on representative graphs. We report the running times of all algorithms on 30 representative graphs with $k = 5$ in Table 3. We observe that kPEX outperforms all baselines by achieving significant speedups on the majority graphs. For example, kPEX runs at least 5 times faster than KPLEX on 25 out of 30 graphs and at least 5 times faster than kPlexT on 21 out of 30 graphs. Moreover, there are 7 out of 30 graphs where kPEX runs at least 100 times faster than all baselines. Note that kPEX may exhibit slower performance compared to baselines on rare occasions. For instance, kPlexT runs faster than kPEX on G23 with $k = 5$. The possible reasons are as follows. First, both kPEX and kPlexT find the result after the pre-processing procedure, as verified by lb in Table 6. Thus, they have the running time dominated by the pre-processing method. Second, our pre-processing methods (i.e., CF-CTCP and KPHeuris) have the time complexity of $O(nm)$, and thus they cost more than the existing one with the time complexity of $O(\delta m)$.

Memory usage on representative graphs. We evaluate the space consumption by measuring the peak memory usage in Table 4. We can observe that our kPEX achieves comparable memory usage compared to the state-of-the-art algorithm kPlexT, which is also aligned with our theoretical analysis.

6.2 Effectiveness of Proposed Techniques

We compare the running time of kPEX with its variants:

- **kPEX-SeqRB**: kPEX replaces AltRB with SeqRB (Section 4).
- **kPEX-CTCP**: kPEX replaces CF-CTCP with CTCP ([11]).

Table 3: Running time in seconds of kPEX and state-of-the-arts on 30 graphs with $k = 5$

ID	kPEX (ours)	KPLEX	kPlexT	kPlexS	DiseMKP
G1	**4.02**	1154.76	118.88	186.90	53.21
G2	**1324.85**	OOT	OOT	OOT	OOT
G3	**1485.70**	OOT	OOT	OOT	OOT
G4	**297.49**	OOT	3063.87	OOT	OOT
G5	0.14	0.29	34.64	1369.22	**0.13**
G6	**24.11**	OOT	OOT	OOT	OOT
G7	**421.54**	OOT	OOT	OOT	OOT
G8	**2.17**	567.84	20.29	OOT	26.83
G9	**2.75**	411.26	OOT	OOT	OOT
G10	**186.24**	OOT	1291.16	OOT	1098.71
G11	**4.02**	1766.68	2295.12	OOT	OOT
G12	**0.49**	1.30	0.66	1.54	426.92
G13	**53.98**	OOT	OOT	OOT	OOT
G14	**901.50**	OOT	OOT	OOT	OOT
G15	**20.89**	OOT	OOT	OOT	OOT
G16	**1.81**	58.82	26.94	1659.47	870.09
G17	**1.63**	3022.10	122.12	OOT	OOT
G18	**0.86**	2804.46	1721.30	1088.71	OOT
G19	**0.82**	3.59	2.28	1.98	1639.76
G20	**4.32**	795.20	205.57	73.81	1296.59
G21	**3.00**	961.06	1491.24	OOT	OOT
G22	**2.43**	13.58	3.10	13.88	16.20
G23	12.89	136.61	**4.84**	9.57	OOT
G24	**5.56**	2979.37	3079.64	OOT	OOT
G25	**3.63**	92.05	199.73	OOT	OOT
G26	**4.87**	700.66	6.84	40.58	8.49
G27	**2.43**	14.52	4.43	16.73	48.41
G28	**132.41**	OOT	OOT	2078.00	OOT
G29	**6.01**	312.93	OOT	OOT	OOT
G30	**589.81**	OOT	OOT	OOT	OOT

- **kPEX-EGo**: kPEX replaces KPHeuris with the existing heuristic method EGo-Degen in [12].
- **kPEX-Degen**: kPEX replaces KPHeuris with the existing heuristic method kPlex-Degen in [11].

Effectiveness of AltRB. We compare kPEX with kPEX-SeqRB in Table 5. We observe that kPEX outperforms kPEX-SeqRB by achieving at least a 5× speedup on 12 out of 30 graphs and up to 20× faster on G6. This indicates the effectiveness of AltRB in narrowing down the search space. Besides, AltRB contributes more speedups on synthetic graphs G1-G10 since the running time is dominated by the branch-reduction-and-bound stage on these graphs.

Effectiveness of CF-CTCP. We compare kPEX with kPEX-CTCP, and the running times are reported in Table 5. First, kPEX and kPEX-CTCP have similar performance on G1-G10 because the pre-processing techniques take little time (e.g., less than 1 second) on these synthetic graphs. Second, kPEX runs at least 5 times faster than kPEX-CTCP on 8 out of 20 real-world graphs. Moreover, CF-CTCP provides at least 50× speedup on G20 and G23. These results show the effectiveness of CF-CTCP on large sparse graphs.

Effectiveness of KPHeuris. We compare kPEX with its variants kPEX-EGo and kPEX-Degen (note that CF-CTCP is not replaced). The running times are shown in Table 5. We have the following observations. First, the running time of kPEX is less than that of both variants on the majority of graphs (i.e., on 24 out of 30 graphs). Then, kPEX runs at least 5 times faster than kPEX-EGo on 5 out of

Table 4: Peak memory usage (MB) of kPEX and state-of-the-arts on 30 graphs with $k = 5$

ID	kPEX (ours)	KPLEX	kPlexT	kPlexS	DiseMKP
G1	4.05	4.37	**3.92**	3.95	309.02
G2	4.08	3.96	**3.90**	4.18	309.03
G3	4.45	4.25	**3.86**	4.38	308.76
G4	4.41	4.53	4.41	**4.35**	308.75
G5	4.97	10.10	**4.26**	5.26	309.14
G6	4.95	4.61	**4.20**	5.02	309.01
G7	5.01	4.62	**4.25**	4.88	309.11
G8	4.60	41.90	**4.26**	4.62	309.14
G9	5.07	39.50	**4.23**	4.99	309.27
G10	5.72	27.31	**4.61**	6.27	309.68
G11	**8.41**	171.76	11.37	13.45	317.16
G12	**22.70**	45.81	28.28	45.74	345.92
G13	**27.78**	460.00	35.45	54.79	353.96
G14	**32.70**	415.21	52.61	58.65	360.40
G15	**41.18**	415.13	50.96	77.96	376.38
G16	**38.81**	116.17	46.17	79.57	374.84
G17	**33.26**	142.67	35.39	62.44	363.25
G18	**19.48**	43.94	19.90	22.15	332.86
G19	**49.07**	66.39	49.93	341.85	374.57
G20	**207.84**	667.92	410.23	498.40	703.68
G21	58.50	270.51	**49.22**	51.53	368.32
G22	**317.62**	582.74	332.85	571.60	872.73
G23	**359.59**	739.18	395.00	699.88	889.48
G24	108.00	493.18	**94.93**	98.12	427.63
G25	**352.41**	568.52	363.24	492.09	890.86
G26	758.79	834.56	**755.30**	762.48	1555.66
G27	**363.19**	599.40	372.87	586.65	917.98
G28	**1836.75**	3891.50	2312.38	15724.75	3523.29
G29	1455.82	1430.20	**1336.76**	1430.23	2049.30
G30	**4434.79**	6572.04	4755.56	6115.37	6827.20

30 graphs and faster than kPEX-Degen on 4 out of 30 graphs. In addition, kPEX runs at least 25 times faster than both kPEX-EGo and kPEX-Degen on G20 and G28. This shows that making more effort to finding a larger initial k-plex benefits kPEX by narrowing down the search space. Second, although kPEX may be slightly slower than the two variants, the extra time consumption is small and can be ignored compared to the total running time. For example, kPEX is 0.15 seconds slower than kPEX-Degen on G11 due to the extra computation, while the total running time of kPEX is 4.02 seconds, which means that the extra time consumption is negligible. Third, the performance of kPEX-EGo and kPEX-Degen is better than kPEX-SeqRB on G1-G10. This means that the variant of kPEX without AltRB is slower than the variant without KPHeuris. This indicates that AltRB provides a greater performance boost than heuristic techniques on those graphs where branch-reduction-and-bound stage dominates the running time.

Effectiveness of KPHeuris and CF-CTCP. We also compare the total pre-processing time and the size of the k-plex (i.e., lb) obtained by different heuristic methods in kPEX, kPlexT, kPlexS, and DiseMKP (note that KPLEX uses the same pre-processing method as kPlexS). The results are reported in Table 6. Note that we exclude the results on synthetic graphs G1-G10 since they have only hundreds of vertices and can be handled within 1 second by all methods. We have the following observations. First, kPEX consistently obtains the largest lb (or matches the largest obtained by others) while the pre-processing time remains comparable to other algorithms. Second, KPHeuris outperforms the other pre-processing algorithms by

Table 5: Running time in seconds of kPEX and its variants on 30 graphs with $k = 5$

ID	kPEX	kPEX-SeqRB	kPEX-CTCP	kPEX-EGo	kPEX-Degen
G1	4.02	18.66	**3.89**	3.91	3.92
G2	**1324.85**	OOT	1364.70	1371.95	1365.94
G3	**1485.70**	OOT	1525.00	1539.98	1538.84
G4	**297.49**	3220.90	305.95	307.76	307.69
G5	0.14	0.11	0.10	0.10	**0.09**
G6	**24.11**	557.07	24.63	33.37	57.68
G7	**421.54**	OOT	434.65	528.89	674.42
G8	**2.17**	18.54	2.22	2.19	2.20
G9	**2.75**	20.47	2.81	4.21	4.18
G10	**186.24**	3472.65	191.72	202.50	202.49
G11	4.02	23.51	4.63	4.19	**3.87**
G12	0.49	**0.39**	2.24	0.88	0.71
G13	**53.98**	555.36	59.38	97.39	103.23
G14	**901.50**	OOT	936.52	1113.52	1197.87
G15	**20.89**	128.27	35.76	29.85	24.84
G16	**1.81**	2.19	5.84	2.35	2.18
G17	**1.63**	8.49	8.27	3.88	1.90
G18	0.86	6.76	1.34	0.90	**0.66**
G19	0.82	**0.66**	17.41	8.36	93.70
G20	**4.32**	4.46	222.02	150.66	260.41
G21	**3.00**	8.03	3.55	3.68	3.32
G22	2.43	**2.26**	16.82	8.28	2.40
G23	12.89	**12.62**	1618.38	461.19	117.33
G24	**5.56**	15.80	6.97	9.83	9.03
G25	3.63	**3.56**	66.56	22.87	3.69
G26	4.87	4.77	4.64	5.61	**4.54**
G27	**2.43**	2.48	24.83	11.07	2.63
G28	**132.41**	139.96	OOT	OOT	OOT
G29	6.01	17.57	**5.99**	11.18	10.85
G30	**589.81**	OOT	1628.00	2094.29	2020.66

Table 6: Pre-processing time in seconds on 20 graphs with k=5 (lb denotes the size of the computed heuristic k-plex)

ID	kPEX		kPlexT		kPlexS		DiseMKP	
	time	lb	time	lb	time	lb	time	lb
G11	0.55	**51**	0.38	50	**0.37**	50	0.70	49
G12	**0.49**	**73**	0.66	69	1.44	34	1.82	34
G13	4.37	**39**	**1.70**	37	2.21	36	3.97	35
G14	9.18	**70**	4.72	67	**2.71**	64	6.40	66
G15	3.30	**49**	**1.98**	48	4.09	48	10.86	47
G16	1.44	**27**	**0.85**	26	1.38	14	3.79	14
G17	**0.66**	**39**	0.74	37	1.79	37	5.51	37
G18	**0.18**	**87**	0.43	**87**	0.33	**87**	0.65	**87**
G19	**0.81**	**44**	2.09	42	0.98	37	9.67	37
G20	**3.44**	**21**	31.30	19	27.54	10	112.59	10
G21	0.82	**44**	**0.45**	43	0.51	43	0.79	42
G22	**2.42**	**34**	3.09	32	13.84	26	14.51	27
G23	12.89	**11**	**4.84**	**11**	9.05	10	1602.40	10
G24	1.38	**44**	**0.62**	41	1.08	41	1.59	41
G25	**2.98**	**77**	6.41	76	25.24	76	47.80	**77**
G26	3.03	**881**	**2.98**	**881**	3.81	**881**	3.78	880
G27	**2.43**	**37**	4.40	35	16.69	32	20.23	33
G28	**96.92**	**17**	685.11	15	1205.12	12	OOT	-
G29	**2.76**	**296**	2.83	292	2.76	292	8.86	292
G30	**94.28**	**65**	107.72	62	179.92	17	1213.23	16

Table 7: Running time in seconds of variants of kPEX using different partition methods on 30 representative graphs

ID	kPEX	dp	rp	ID	kPEX	dp	rp
G1	**4.02**	48.55	53.39	G16	**1.81**	3.48	3.59
G2	**1324.85**	OOT	OOT	G17	**1.63**	20.45	22.45
G3	**1485.70**	OOT	OOT	G18	**0.86**	11.41	13.58
G4	**297.49**	OOT	OOT	G19	0.82	**0.81**	0.81
G5	**0.14**	0.16	0.16	G20	**4.32**	4.75	4.95
G6	**24.11**	2077.73	2380.12	G21	**3.00**	21.69	23.38
G7	**421.54**	OOT	OOT	G22	2.43	2.34	**2.31**
G8	**2.17**	65.11	71.29	G23	12.89	13.00	**12.50**
G9	**2.75**	71.87	78.53	G24	**5.56**	43.72	47.92
G10	**186.24**	OOT	OOT	G25	3.63	3.69	**3.62**
G11	**4.02**	62.93	68.75	G26	4.87	4.96	**4.73**
G12	**0.49**	0.49	0.50	G27	**2.43**	2.54	2.52
G13	**53.98**	1916.44	2155.19	G28	132.41	**131.37**	132.43
G14	**901.50**	OOT	OOT	G29	**6.01**	24.13	25.64
G15	**20.89**	371.52	415.26	G30	**589.81**	OOT	OOT

obtaining a lager k-plex while costing much less time on G20 and G28. This also verifies the effectiveness of CF-CTCP and KPHeuris.

Effectiveness of partition methods. We evaluate the effectiveness of our method Partition (Algortihm 3) by comparing kPEX with its variants. In specific, dp is the degree-based partition which sorts the vertices in S (resp. C) based on their degrees and partitions S (resp. C) into two subsets with the same size; and rp is the random partition which randomly partitions S (resp. C) into two subsets. The running time on 30 representative graphs when $k = 5$ are reported in Table 7. We find that kPEX with Partition outperforms others by achieving at least 10x speedup on 11 out of 30 graphs. This is also consistent with the theoretical analysis given in Equation (9) that AltRB with Partition further narrows down the search space in theory.

7 RELATED WORK

Maximum k-plex search. The *maximum k-plex search* problem has garnered significant attention in social network analysis [42, 43] since the concept of k-plex was first proposed in [48]. Balasundaram et al. [3] showed the NP-hardness of the problem with any fixed k. Consequently, the major algorithmic design paradigm for exact solution is based on the *branch-reduction-and-bound* (BRB) framework [11, 12, 27, 35, 36, 53, 56, 63]. In particular, Xiao et al. [56] proposed a new branching strategy. Later, Wang et al. [53] designed **KPLEX** which is parameterized by the degeneracy gap (bounded empirically by $O(\log n)$). Very recently, Chang and Yao [12] proposed **kPlexT** with newly proposed branching and reduction techniques. Additionally, several reduction and bounding techniques have been designed in the BRB framework to boost the practical performance. Gao et al. [27] developed reduction methods and a dynamic vertex selection strategy. Later, Zhou et al. [63] proposed a stronger reduction method and designed a coloring-based bounding method. Jiang et al. [36] designed a partition-based bounding method, and later in [35], their algorithm **DiseMKP** is equipped with a better upper bound. Chang et al. [11] designed an efficient algorithm **kPlexS** with a novel reduction method CTCP and a heuristic method. We note that the algorithms designed by Xiao et al. [56] and Chang and Yao [12] also work for the case when there is no requirement for the found k-plex to be of size at least $2k - 1$. We remark that existing works mainly focus on the BRB framework that conducts the reduction and the bounding sequentially, and our solution kPEX firstly adopts a new BRB framework that alternatively and iteratively conducts the reduction and the bounding. Besides, we note that quite a few algorithms are proposed towards achieving better worst-case time complexity. In specific, the time complexity has been improved from $O^*(2^n)$, to $O^*(\beta_k^n)$ [56], to $O^*((k+1)^{\delta+k-|S^*|})$ [53] and $O^*(\gamma_k^\delta)$ [12], where O^* suppresses the polynomial factors and $\gamma_k < \beta_k < 2$. Among them, KPLEX [53] and kPlexT [12] achieve the best. We note that (1) previous SOTA algorithms are all BRB methods and the improvements of time complexity come from different branching strategies, which are orthogonal to our techniques; and (2) the practical performance mainly depends on the later. Hence, by adopting the branching strategies in KPLEX and kPlexT, our kPEX can achieve the same complexity of $O^*((k+1)^{\delta+k-|S^*|})$ and $O^*(\gamma_k^\delta)$, respectively. We also conduct experiments for various branching strategies and find that our kPEX with different branching strategies achieve comparable performance (details can be found in the technical report [29]).

Maximal k-plex enumeration. A related problem is *maximal k-plex enumeration*, which aims to list all maximal k-plexes in the input graph. Here, a k-plex is *maximal* if it cannot be contained in other k-plexes. Efficient algorithms are proposed for enumerating maximal k-plexes, including Bron-Kerbosch-based algorithms [17, 19, 22, 52, 54, 55] and reverse-search-based algorithms [6]. We note that existing algorithms for enumerating maximal k-plexes can be used to solve our studied problem by listing all maximal k-plexes and then returning the largest one among them (note that the maximum k-plex is the maximal k-plex with largest number of vertices). However, the resulting solutions are not efficient due to the limited pruning and bounding techniques, as verified in [12].

Other cohesive subgraph models. k-plexes reduce to cliques when $k = 1$. There exist numerous studies focusing on the maximum clique search and maximal clique enumeration problems [7, 8, 18, 24, 44, 45, 49, 50]. Further, the concept of k-plex is also explored in other kinds of graphs, e.g., bipartite graphs [13, 23, 40, 59, 60], directed graphs [30], temporal graphs [5], uncertain graphs [20], and so on. Besides k-plex, various cohesive subgraph models have been studied, including k-core [4, 15], k-truss [16, 33, 51], γ-quasi-clique [37, 46, 58, 61], k-defective clique [9, 14, 21, 28], densest subgraph [41, 57], and so on. For an overview on cohesive subgraph search, we refer to excellent books and surveys [10, 25, 26, 34, 39].

8 CONCLUSION

In this paper, we studied the maximum k-plex problem. We proposed a new branch-reduction-and-bound method, called kPEX, which includes a new alternated reduction-and-bound process AltRB. We also designed efficient pre-processing techniques for boosting the practical performance, which includes KPHeuris for computing a large initial heuristic k-plex and CF-CTCP for efficiently removing unpromising vertices and edges in the graph. Extensive experiments on 664 graphs verified kPEX's superiority over existing state-of-the-art algorithms. In the future, we will explore the possibility of adapting kPEX to mining other cohesive subgraphs.

REFERENCES

[1] 2nd-DIMACS. http://archive.dimacs.rutgers.edu/pub/challenge/graph/.

[2] Network Repository. https://networkrepository.com/index.php.

[3] Balabhaskar Balasundaram, Sergiy Butenko, and Illya V. Hicks. 2011. Clique Relaxations in Social Network Analysis: The Maximum k-Plex Problem. *Operations Research* 59, 1 (2011), 133–142.

[4] Vladimir Batagelj and Matjaž Zaveršnik. 2003. An $O(m)$ Algorithm for Cores Decomposition of Networks. *CoRR* cs.DS/0310049 (2003).

[5] Matthias Bentert, Anne-Sophie Himmel, Hendrik Molter, Marco Morik, Rolf Niedermeier, and René Saitenmacher. 2019. Listing All Maximal k-Plexes in Temporal Graphs. *ACM J. Exp. Algorithmics* 24, Article 1.13 (Sep 2019).

[6] Devora Berlowitz, Sara Cohen, and Benny Kimelfeld. 2015. Efficient Enumeration of Maximal k-Plexes. In *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*. 431–444.

[7] Randy Carraghan and Panos M. Pardalos. 1990. An Exact Algorithm for the Maximum Clique Problem. *Operations Research Letter* 9, 6 (1990), 375–382.

[8] Lijun Chang. 2019. Efficient Maximum Clique Computation over Large Sparse Graphs. In *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining (SIGKDD)*. 529–538.

[9] Lijun Chang. 2023. Efficient Maximum k-Defective Clique Computation with Improved Time Complexity. *Proceedings of the ACM on Management of Data (SIGMOD)* 1, 3 (2023), 1–26.

[10] Lijun Chang and Lu Qin. 2018. *Cohesive Subgraph Computation over Large Sparse Graphs*. Springer.

[11] Lijun Chang, Mouyi Xu, and Darren Strash. 2022. Efficient Maximum k-Plex Computation over Large Sparse Graphs. *Proceedings of the VLDB Endowment* 16, 2 (2022), 127–139.

[12] Lijun Chang and Kai Yao. 2024. Maximum k-Plex Computation: Theory and Practice. *Proceedings of the ACM on Management of Data (SIGMOD)* 2, 1 (2024), 1–26.

[13] Lu Chen, Chengfei Liu, Rui Zhou, Jiajie Xu, and Jianxin Li. 2021. Efficient Exact Algorithms for Maximum Balanced Biclique Search in Bipartite Graphs. In *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*. 248–260.

[14] Xiaoyu Chen, Yi Zhou, Jin-Kao Hao, and Mingyu Xiao. 2021. Computing Maximum k-Defective Cliques in Massive Graphs. *Computers & Operations Research* 127 (2021), 105131.

[15] James Cheng, Yiping Ke, Shumo Chu, and M Tamer Özsu. 2011. Efficient Core Decomposition in Massive Networks. In *Proceedings of the IEEE International Conference Data Engineering (ICDE)*. 51–62.

[16] Jonathan Cohen. 2008. Trusses: Cohesive Subgraphs for Social Network Analysis. *National Security Agency Technical Report* 16, 3.1 (2008).

[17] Alessio Conte, Tiziano De Matteis, Daniele De Sensi, Roberto Grossi, Andrea Marino, and Luca Versari. 2018. D2K: Scalable Community Detection in Massive Networks via Small-Diameter k-Plexes. In *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining (SIGKDD)*. 1272–1281.

[18] Alessio Conte, Roberto De Virgilio, Antonio Maccioni, Maurizio Patrignani, Riccardo Torlone, et al. 2016. Finding All Maximal Cliques in Very Large Social Networks. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*. 173–184.

[19] Alessio Conte, Donatella Firmani, Caterina Mordente, Maurizio Patrignani, and Riccardo Torlone. 2017. Fast Enumeration of Large k-Plexes. In *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining (SIGKDD)*. 115–124.

[20] Qiangqiang Dai, Rong-Hua Li, Meihao Liao, Hongzhi Chen, and Guoren Wang. 2022. Fast Maximal Clique Enumeration on Uncertain Graphs: A Pivot-based Approach. In *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*. 2034–2047.

[21] Qiangqiang Dai, Rong-Hua Li, Meihao Liao, and Guoren Wang. 2023. Maximal Defective Clique Enumeration. *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)* 1, 1 (2023), 1–26.

[22] Qiangqiang Dai, Rong-Hua Li, Hongchao Qin, Meihao Liao, and Guoren Wang. 2022. Scaling Up Maximal k-Plex Enumeration. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM)*. 345–354.

[23] Qiangqiang Dai, Rong-Hua Li, Xiaowei Ye, Meihao Liao, Weipeng Zhang, and Guoren Wang. 2023. Hereditary Cohesive Subgraphs Enumeration on Bipartite Graphs: The Power of Pivot-based Approaches. *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)* 1, 2 (2023), 1–26.

[24] David Eppstein, Maarten Löffler, and Darren Strash. 2013. Listing All Maximal Cliques in Large Sparse Real-World Graphs. *ACM J. Exp. Algorithmics* 18, Article 3.1 (Nov 2013).

[25] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. 2020. A Survey of Community Search over Big Graphs. *The VLDB Journal* 29 (2020), 353–392.

[26] Yixiang Fang, Kai Wang, Xuemin Lin, and Wenjie Zhang. 2021. Cohesive Subgraph Search over Big Heterogeneous Information Networks: Applications, Challenges, and Solutions. In *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*. 2829–2838.

[27] Jian Gao, Jiejiang Chen, Minghao Yin, Rong Chen, and Yiyuan Wang. 2018. An Exact Algorithm for Maximum k-Plexes in Massive Graphs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 1449–1455.

[28] Jian Gao, Zhenghang Xu, Ruizhi Li, and Minghao Yin. 2022. An Exact Algorithm with New Upper Bounds for the Maximum k-Defective Clique Problem in Massive Sparse Graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 10174–10183.

[29] Shuohao Gao, Kaiqiang Yu, Shengxin Liu, and Cheng Long. 2024. Maximum k-Plex Search: An Alternated Reduction-and-Bound Method (Technical report). https://shuohaogao.github.io/pdf/MaxKPlex_AltRB-full.pdf.

[30] Shuohao Gao, Kaiqiang Yu, Shengxin Liu, Cheng Long, and Zelong Qiu. 2024. On Searching Maximum Directed (k, ℓ)-Plex. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 2570–2583.

[31] Guimu Guo, Da Yan, M Tamer Özsu, Zhe Jiang, and Jalal Khalil. 2020. Scalable Mining of Maximal Quasi-Cliques: An Algorithm-System Codesign Approach. *Proceedings of the VLDB Endowment* 14, 4 (2020), 573–585.

[32] Guimu Guo, Da Yan, Lyuheng Yuan, Jalal Khalil, Cheng Long, Zhe Jiang, and Yang Zhou. 2022. Maximal Directed Quasi-Clique Mining. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 1900–1913.

[33] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-Truss Community in Large and Dynamic graphs. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 1311–1322.

[34] Xin Huang, Laks V. S. Lakshmanan, and Jianliang Xu. 2019. *Community Search over Big Graphs*. Morgan & Claypool Publishers.

[35] Hua Jiang, Fusheng Xu, Zhifei Zheng, Bowen Wang, and Wei Zhou. 2023. A Refined Upper Bound and Inprocessing for the Maximum k-Plex Problem. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 5613–5621.

[36] Hua Jiang, Dongming Zhu, Zhichao Xie, Shaowen Yao, and Zhang-Hua Fu. 2021. A New Upper Bound Based on Vertex Partitioning for the Maximum k-Plex Problem. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 1689–1696.

[37] Jalal Khalil, Da Yan, Guimu Guo, and Lyuheng Yuan. 2022. Parallel Mining of Large Maximal Quasi-Cliques. *The VLDB Journal* 31, 4 (2022), 649–674.

[38] Valdis E Krebs. 2002. Mapping Networks of Terrorist Cells. *Connections* 24, 3 (2002), 43–52.

[39] Victor E. Lee, Ning Ruan, Ruoming Jin, and Charu Aggarwal. 2010. A Survey of Algorithms for Dense Subgraph Discovery. *Managing and Mining Graph Data* (2010), 303–336.

[40] Wensheng Luo, Kenli Li, Xu Zhou, Yunjun Gao, and Keqin Li. 2022. Maximum Biplex Search over Bipartite Graphs. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 898–910.

[41] Chenhao Ma, Yixiang Fang, Reynold Cheng, Laks VS Lakshmanan, Wenjie Zhang, and Xuemin Lin. 2021. On Directed Densest Subgraph Discovery. *ACM Transactions on Database Systems (TODS)* 46, 4 (2021), 1–45.

[42] Benjamin McClosky and Illya V. Hicks. 2012. Combinatorial Algorithms for the Maximum k-Plex Problem. *Journal of Combinatorial Optimization* 23, 1 (2012), 29–49.

[43] Hannes Moser, Rolf Niedermeier, and Manuel Sorge. 2012. Exact Combinatorial Algorithms and Experiments for Finding Maximum k-Plexes. *Journal of Combinatorial Optimization* 24, 3 (2012), 347–373.

[44] Kevin A. Naudé. 2016. Refined Pivot Selection for Maximal Clique Enumeration in Graphs. *Theoretical Computer Science* 613 (2016), 28–37.

[45] Panos M. Pardalos and Jue Xue. 1994. The Maximum Clique Problem. *Journal of Global Optimization* 4, 3 (1994), 301–328.

[46] Jian Pei, Daxin Jiang, and Aidong Zhang. 2005. On Mining Cross-Graph Quasi-Cliques. In *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining (SIGKDD)*. 228–238.

[47] Stephen B. Seidman. 1983. Network Structure and Minimum Degree. *Social Networks* 5, 3 (1983), 269–287.

[48] Stephen B. Seidman and Brian L. Foster. 1978. A Graph-Theoretic Generalization of the Clique Concept. *Journal of Mathematical Sociology* 6, 1 (1978), 139–154.

[49] Etsuji Tomita. 2017. Efficient Algorithms for Finding Maximum and Maximal Cliques and Their Applications. In *Proceedings of the International Conference and Workshops on Algorithms and Computation (WALCOM)*. 3–15.

[50] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. 2006. The Worst-Case Time Complexity for Generating All Maximal Cliques and Computational Experiments. *Theoretical Computer Science* 363, 1 (2006), 28–42.

[51] Jia Wang and James Cheng. 2012. Truss Decomposition in Massive Networks. *Proceedings of the VLDB Endowment* 5, 9 (2012), 812–823.

[52] Zhuo Wang, Qun Chen, Boyi Hou, Bo Suo, Zhanhuai Li, Wei Pan, and Zachary G. Ives. 2017. Parallelizing Maximal Clique and k-Plex Enumeration over Graph Data. *J. Parallel and Distrib. Comput.* 106 (2017), 79–91.

[53] Zhengren Wang, Yi Zhou, Chunyu Luo, and Mingyu Xiao. 2023. A Fast Maximum k-Plex Algorithm Parameterized by the Degeneracy Gap. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*. 5648–5656.

[54] Zhengren Wang, Yi Zhou, Mingyu Xiao, and Bakhadyr Khoussainov. 2022. Listing Maximal k-Plexes in Large Real-World Graphs. In *Proceedings of the ACM Web Conference (WWW)*. 1517–1527.

[55] Bin Wu and Xin Pei. 2007. A Parallel Algorithm for Enumerating All the Maximal k-Plexes. In *Proceedings of the International Workshops on Emerging Technologies in Knowledge Discovery and Data Mining (PAKDD workshop)*. 476–483.
[56] Mingyu Xiao, Weibo Lin, Yuanshun Dai, and Yifeng Zeng. 2017. A Fast Algorithm to Compute Maximum k-Plexes in Social Network Analysis. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 919–925.
[57] Yichen Xu, Chenhao Ma, Yixiang Fang, and Zhifeng Bao. 2024. Efficient and Effective Algorithms for Densest Subgraph Discovery and Maintenance. *The VLDB Journal* (2024).
[58] Kaiqiang Yu and Cheng Long. 2023. Fast Maximal Quasi-Clique Enumeration: A Pruning and Branching Co-Design Approach. *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)* 1, 3 (2023), 1–26.
[59] Kaiqiang Yu and Cheng Long. 2023. Maximum k-Biplex Search on Bipartite Graphs: A Symmetric-BK Branching Approach. *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)* 1, 1 (2023), 1–26.
[60] Kaiqiang Yu, Cheng Long, Shengxin Liu, and Da Yan. 2022. Efficient Algorithms for Maximal k-Biplex Enumeration. In *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*. 860–873.
[61] Zhiping Zeng, Jianyong Wang, Lizhu Zhou, and George Karypis. 2006. Coherent Closed Quasi-Clique Discovery from Large Dense Graph Databases. In *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining (SIGKDD)*. 797–802.
[62] Yun Zhang, Charles A Phillips, Gary L Rogers, Erich J Baker, Elissa J Chesler, and Michael A Langston. 2014. On Finding Bicliques in Bipartite Graphs: A Novel Algorithm and its Application to the Integration of Diverse Biological Data Types. *BMC Bioinformatics* 15 (2014), 1–18.
[63] Yi Zhou, Shan Hu, Mingyu Xiao, and Zhang-Hua Fu. 2021. Improving Maximum k-Plex Solver via Second-Order Reduction and Graph Color Bounding. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*. 12453–12460.

A ADDITIONAL DESCRIPTIONS OF CF-CTCP

A.1 Time Complexity of CF-CTCP

Before analyzing the time complexity of CF-CTCP (Algorithm 4), we first prove the following lemma.

Lemma A.1. *Given a graph $G = (V, E)$, we have*

$$\sum_{(u,v)\in E} \min(d_G(u), d_G(v)) \leq 2m \times \delta(G).$$

Proof. Assume that vertices $v_1, v_2, ..., v_n$ in G are sorted according to the degeneracy order, indicating that $|N_G^+(v_i)| = |N_G(v_i) \cap \{v_{i+1}, v_{i+2}, ..., v_n\}| \leq \delta(G)$. Thus we have

$$\begin{aligned}
&\sum_{(u,v)\in E} \min(d_G(u), d_G(v)) \\
&= \sum_{v_i\in V} \sum_{v_j \in N_G^+(v_i)} \min(d_G(v_i), d_G(v_j)) \\
&\leq \sum_{v_i\in V} \sum_{v_j \in N_G^+(v_i)} d_G(v_i) \leq \sum_{v_i\in V} d_G(v_i) \times \delta(G) = 2m \times \delta(G).
\end{aligned}$$

□

We can derive from Lemma A.1 that

$$O(\sum_{(u,v)\in E} \min(d_G(u), d_G(v))) = O(m \times \delta(G)).$$

Now we are ready to prove the total time complexity of CF-CTCP (Lemma 5.3).

Proof. Note that we invoke CF-CTCP only when $Q_v \neq \emptyset$ or $lb\_changed = true$ as in CTCP [11]. First, for the first invocation, Line 6 of Algorithm 4 computes the common neighbors $\Delta(u, v)$ for each edge (u, v), and the time complexity is

$$O(\sum_{(u,v)\in E} \min(d_G(u), d_G(v))) = O(m \times \delta(G)),$$

according to Lemma A.1. Second, the total time consumption of core pruning is $O(m)$ [4] and the total time cost of Procedure RemoveEdge is also $O(m)$ since we can implement Line 21 for at most m times. Third, for all invocations, there are at most $\delta(G)$ times when $lb\_changed = true$ since $k \leq lb = |S^*| \leq \delta(G) + k$ (S^* denoting the largest k-plex seen so far), which indicates that we will perform Lines 4-8 at most $\delta(G)$ times. Thus the total time complexity of Lines 1-8 is $O(m \times \delta(G))$. We next consider Lines 9-20. We will pop at most m edges, and for each edge, we need to find all the triangles that it participates in, which can be done in $O(\sum_{(u,v)\in E} \min(d_G(u), d_G(v))) = O(m \times \delta(G))$. Therefore, the total time complexity of all invocations to CF-CTCP is $O(m \times \delta(G))$, which completes our proof. □

A.2 An Implementation of CF-CTCP with $O(m)$ Memory

A direct implementation of CF-CTCP requires storing the common neighbors $\Delta(\cdot, \cdot)$ for all edges, which needs $O(m \times \delta(G))$ memory. In the following, we propose a novel implementation that requires only $O(m)$ memory without changing the time complexity of CF-CTCP. In particular, we need three auxiliary arrays A_1, A_2, and A_3, each of length m, to store additional information for each edge: 1) array A_1 records the number of triangles, 2) array A_2 records the timestamp (e.g., system time) when the triangle count is computed in Line 6, and 3) array A_3 records the timestamp (e.g., system time) when an edge is removed in Lines 1, 21 and 22. Based on these three arrays, we correspondingly modify Algorithm 4 as follows. First, we only record $|\Delta(u, v)|$ using A_1 instead of storing the whole vertex set $\Delta(u, v)$ in Line 6. The correspond triangle count in A_1 is decreased by 1 when CF-CTCP modifies $\Delta(\cdot, \cdot)$ in Lines 13 and 18. Second, when we traverse all triangles that edge (u, v) belongs to in Line 12, we enumerate such a vertex w that satisfies: 1) both (u, w) and (v, w) are in $E \cup Q_e$, i.e., (u, v, w) forms a triangle; 2) the timestamp of computing the triangle count for edge (u, w) is before the timestamp of removing edge (u, v) using arrays A_2 and A_3, i.e., when we compute $|\Delta(u, w)|$ in Line 6, edge (u, v) has not yet been removed. The modification to Line 17 follows the same fashion as Line 12. Finally, it is easy to verify the correctness of the above modification of CF-CTCP with $O(m)$ memory usage.

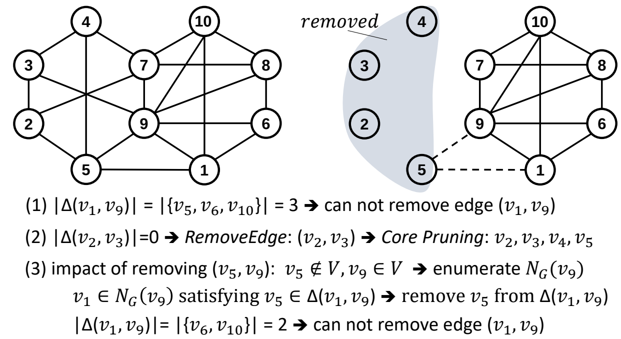


Figure 4: An example for CF-CTCP assuming $lb = 4$ and $k = 2$

A.3 An Example of CF-CTCP

Consider the example of CF-CTCP (Algorithm 4) in Figure 4, assuming $lb = 4$ and $k = 2$. According to Lemma 5.1 and Lemma 5.2, we need to reduce G to the maximal subgraph that is both a 3-core and a 3-truss, i.e., we will remove a vertex u if $d_G(u) < 3$ and an edge (u, v) if $|\Delta(u, v)| < 1$. First, we enumerate each edge (u, v) and compute the common neighbors of u and v (Lines 4-6). For those edges connected to v_1, we cannot remove them since they have enough common neighbors, e.g., there are 3 common neighbors of v_1 and v_9. However, when we consider edges connected to v_2, we find that edge (v_2, v_3) can be removed since $|\Delta(v_2, v_3)| = 0$. We then immediately remove edge (v_2, v_3) and conduct core pruning, which removes vertices v_2, v_3, v_4, and v_5 (Lines 21-24). After this process, we continue to compute common neighbors for the remaining edges, but none of these edges can be removed. Second, we begin to consider those removed edges in Q_e. We focus on the edges (v_5, v_9) and (v_1, v_5) since the other removed edges cannot form a triangle with the remaining edges in G. For the removal of the edge (v_5, v_9), according to Lines 11-20, we update $\Delta(v_1, v_9)$, and the triangle (v_1, v_5, v_9) is destroyed. Then, for the edge (v_1, v_5), since the triangle (v_1, v_5, v_9) no longer exists after removing the edge (v_5, v_9), the edge (v_1, v_5) cannot constitute any triangle with other vertices. Thus, the procedure of CF-CTCP terminates. Finally, we

Table 8: Number of solved instances within 3600 seconds when k=5 (an instance is *small* if $|S^*| < 2k-1$; *large*, otherwise. *sum=small+large*.)

Solvers	DIMACS-10			real-world		
	small	large	sum	small	large	sum
kPEX	31	44	**75**	24	114	**138**
kPlexT	25	44	69	23	111	134
kPlexS	22	44	66	24	110	134
KPLEX	19	44	63	21	112	133
DiseMKP	31	35	66	24	102	126
KpLeX	19	35	54	21	101	122

reduce G to $G[\{v_1, v_6, v_7, v_8, v_9, v_{10}\}]$ where $G[\{v_1, v_6, v_8, v_9, v_{10}\}]$ is a 2-plex of size 5.

B ADDITIONAL EXPERIMENTAL RESULTS

We provide additional experimental results including (1) the evaluation of the algorithms for the maximum k-plex search problem without size constraint, and (2) results for omitted values of k when $k = 2, 3, 10, 15, 20$.

B.1 Evaluation for the maximum k-plex problem without size constraint

We compare our algorithm kPEX with existing state-of-the-arts for the maximum k-plex problem without size constraint.

Datasets and Setups. We use the same datasets as in [12], **10th-DIMACS** graphs collection (**DIMACS-10**) and **real-world** graphs collection, which contain 84 graphs and 139 graphs, respectively. The time limit is set as 3600 seconds. Following [12], if KPLEX (resp. kPlexS) cannot find a k-plex of size at least $2k-1$ (resp. $2k-2$), then its running time also includes the running time of another baseline solver KpLeX [36], which is designed without the size constraint. The lower bound initialization of $2k-2$ is disabled for our kPEX so that kPEX is able to find maximum k-plex without size constraint.

The experimental results for k=5 are reported in Table 8. For clear justifications, we distinguish the solved graph instances by (1) *small*: $|S^*| < 2k-1$ and (2) *large*: $|S^*| \geq 2k-1$. We have the following observations. First, our kPEX solves more instances than all competitors on both datasets, which further demonstrates the efficiency of AltRB and two novel pre-processing techniques. Second, when focusing on *large* graphs, KPLEX is comparable with (on **DIMACS-10** graphs) or better (on **real-world** graphs) than kPlexT and kPlexS, which is similar to our observation in Section 6.1. Third, when focusing on *small* graphs, the performance of KPLEX is determined by KpLeX since KPLEX is not able to solve these instances. Thus, kPlexT outperforms KPLEX on *small* graphs. In addition, when kPlexS reports a k-plex of size $2k-2$, its running time does not include the time of KpLeX. Consequently, kPlexS is also slightly better than KPLEX on *small* graphs.

B.2 Comparing with State-of-the-art Algorithms

Running times on representative graphs. We report the running times of all algorithms on 30 representative graphs with k =2, 3, 10, 15, and 20 in Tables 9, 10, 11, 12, and 13, respectively. We observe that kPEX outperforms all baselines by achieving significant speedups on the majority graphs. For example, kPEX runs at least 5 times faster than KPLEX on 23 out of 30 graphs and at least 5 times faster than kPlexT on 20 out of 30 graphs when $k = 3$.

Table 9: Running time in seconds of kPEX and state-of-the-arts on 30 graphs with $k = 2$

ID	kPEX (ours)	KPLEX	kPlexT	kPlexS	DiseMKP
G1	**1.11**	1.77	1.73	2.65	1.39
G2	1932.28	**1847.75**	OOT	OOT	OOT
G3	50.47	105.06	128.54	178.50	**21.25**
G4	**1.47**	9.46	8.10	18.89	2.12
G5	74.91	**27.08**	OOT	OOT	OOT
G6	OOT	OOT	OOT	OOT	OOT
G7	OOT	OOT	OOT	OOT	OOT
G8	0.33	2.60	2.60	18.91	**0.31**
G9	**38.64**	86.99	86.60	637.44	490.15
G10	4.17	33.39	29.82	476.79	**3.18**
G11	**20.57**	127.32	120.10	1004.89	OOT
G12	**0.69**	1.45	0.85	1.31	13.40
G13	**279.91**	2505.66	2768.53	OOT	2321.59
G14	**3082.80**	OOT	OOT	OOT	OOT
G15	**138.28**	1420.48	1591.15	OOT	OOT
G16	**2.05**	7.69	8.04	31.42	7.98
G17	**6.30**	39.70	51.33	219.79	98.73
G18	**2.40**	8.16	68.66	60.86	OOT
G19	**8.81**	11.07	22.47	9.57	OOT
G20	**5.62**	23.69	75.89	18.97	309.47
G21	**13.08**	118.76	122.11	941.83	326.01
G22	**3.18**	16.27	4.34	17.21	20.65
G23	8.61	10.12	**3.93**	8.76	992.05
G24	**24.07**	258.56	241.92	2338.14	532.58
G25	**13.39**	79.16	76.26	273.30	OOT
G26	11.38	41.78	**6.01**	109.56	224.96
G27	**3.35**	19.32	4.48	15.92	27.89
G28	**139.40**	849.77	OOT	1385.86	OOT
G29	**5.92**	7.71	2129.05	531.54	OOT
G30	**136.74**	514.32	701.18	1502.36	OOT

B.3 Effectiveness of Proposed Techniques

Effectiveness of AltRB. We compare kPEX with kPEX-SeqRB and report the running times for k=2, 3, 10, 15, and 20 in Tables 14, 15, 16, 17, and 18, respectively. We observe that kPEX performs better than kPEX-SeqRB at most times, and AltRB can bring at least 60× speedup on G5 when k=10. In addition, we observe that the gap between kPEX and kPEX-AltRB narrows when $k \geq 15$. A possible reason may be that finding a larger heuristic k-plex (i.e., KPHeuris) is more important than AltRB for large values of k.

Effectiveness of CF-CTCP. We compare kPEX with kPEX-CTCP, and the running times for k=2, 3, 10, 15, and 20 are reported in Tables 14, 15, 16, 17, and 18, respectively. First, kPEX and kPEX-CTCP still have similar performance on G1-G10 because the pre-processing techniques take little time on these small synthetic graphs. Second, kPEX runs stably at least 50 times faster than kPEX-CTCP on G23 for all tested values of k. These results show the effectiveness of CF-CTCP on large sparse graphs.

Effectiveness of KPHeuris. We compare kPEX with its variants kPEX-EGo and kPEX-Degen (note that CF-CTCP is not replaced). The running times for k=2, 3, 10, 15, and 20 are shown in Tables 14, 15, 16, 17, and 18, respectively. We have the following

Table 10: Running time in seconds of kPEX and state-of-the-arts on 30 graphs with $k = 3$

ID	kPEX (ours)	KPLEX	kPlexT	kPlexS	DiseMKP
G1	**5.25**	32.14	22.93	22.39	7.11
G2	OOT	OOT	OOT	OOT	OOT
G3	58.89	1112.70	2024.34	1455.21	**21.80**
G4	**9.08**	269.58	67.58	706.34	10.83
G5	0.14	0.46	1.23	15.67	**0.11**
G6	**2.75**	28.08	1551.45	OOT	48.61
G7	OOT	OOT	OOT	OOT	OOT
G8	**0.21**	73.99	2.87	322.75	1.05
G9	**20.59**	528.72	937.24	3597.69	2158.03
G10	14.66	2622.38	160.43	OOT	**10.03**
G11	**6.53**	OOT	367.52	OOT	OOT
G12	**0.53**	1.46	0.77	1.32	15.28
G13	**158.38**	OOT	OOT	OOT	OOT
G14	OOT	OOT	OOT	OOT	OOT
G15	**45.68**	OOT	2673.73	OOT	OOT
G16	**1.58**	167.99	7.57	184.76	19.99
G17	**2.34**	987.10	34.51	925.61	237.93
G18	**1.88**	577.38	279.27	311.55	OOT
G19	1.17	3.36	**0.97**	1.32	2828.96
G20	**4.24**	102.50	26.17	24.29	299.64
G21	**9.00**	OOT	344.24	OOT	1658.25
G22	**2.88**	15.34	3.94	14.19	20.69
G23	12.68	150.23	**4.83**	11.17	1696.51
G24	**14.90**	OOT	634.09	OOT	OOT
G25	**6.87**	2877.27	178.04	1927.66	OOT
G26	**6.85**	368.98	9.98	245.98	49.07
G27	**2.87**	17.59	4.32	14.80	27.22
G28	**109.15**	OOT	OOT	1314.90	OOT
G29	**112.05**	825.59	OOT	OOT	OOT
G30	**189.61**	OOT	OOT	OOT	OOT

Table 11: Running time in seconds of kPEX and state-of-the-arts on 30 graphs with $k = 10$

ID	kPEX (ours)	KPLEX	kPlexT	kPlexS	DiseMKP
G1	**450.15**	OOT	OOT	3104.85	OOT
G2	**37.72**	OOT	OOT	OOT	OOT
G3	OOT	OOT	OOT	OOT	OOT
G4	OOT	OOT	OOT	OOT	OOT
G5	**14.24**	OOT	OOT	OOT	OOT
G6	OOT	OOT	OOT	OOT	OOT
G7	OOT	OOT	OOT	OOT	OOT
G8	OOT	OOT	OOT	OOT	OOT
G9	**301.28**	OOT	OOT	OOT	OOT
G10	OOT	OOT	OOT	OOT	OOT
G11	**6.01**	OOT	OOT	OOT	OOT
G12	**0.65**	1.12	0.73	1.19	OOT
G13	**835.65**	OOT	OOT	OOT	OOT
G14	**3110.98**	OOT	OOT	OOT	OOT
G15	**22.56**	OOT	OOT	OOT	OOT
G16	**2.11**	23.08	OOT	292.88	OOT
G17	**1.79**	314.96	OOT	OOT	OOT
G18	**0.49**	2679.72	OOT	1009.67	OOT
G19	**0.36**	3.04	5.28	2.76	1342.87
G20	**4.90**	409.33	OOT	30.42	OOT
G21	**20.44**	OOT	OOT	OOT	OOT
G22	**2.09**	11.16	3.14	8.90	15.80
G23	12.59	163.83	**4.79**	12.05	OOT
G24	**33.64**	OOT	OOT	OOT	OOT
G25	**2.87**	34.57	OOT	580.38	OOT
G26	**1.18**	3.61	3.74	3.68	3.84
G27	**2.33**	15.41	4.04	11.34	OOT
G28	**260.74**	OOT	OOT	OOT	OOT
G29	**2.82**	101.16	OOT	OOT	10.51
G30	OOT	OOT	OOT	OOT	OOT

observations. First, the running time of kPEX is less than that of both variants on the majority of graphs. Then, kPEX runs up to three orders of magnitude faster than both kPEX-EGo and kPEX-Degen on G19 when k=20. This shows that making more effort to finding a larger initial k-plex benefits kPEX by narrowing down the search space.

Effectiveness of KPHeuris and CF-CTCP. We also compare the total pre-processing time and the size of the k-plex (i.e., lb) obtained by different heuristic methods in kPEX, kPlexT, kPlexS, and DiseMKP (note that KPLEX uses the same pre-processing method as kPlexS). The results for k=2, 3, 10, 15, and 20 are reported in Tables 19, 20, 21, 22, and 23, respectively. Note that we exclude the results on synthetic graphs G1-G10 since they have only hundreds of vertices and can be handled within 1 second by all methods. We have the following observations. First, kPEX obtains the largest lb (or matches the largest obtained by others) at most time while the pre-processing time remains comparable to other algorithms. Second, KPHeuris outperforms the other pre-processing algorithms by obtaining a lager k-plex while costing much less time on G20 and G22 for all tested values of k . This also verifies the effectiveness of CF-CTCP and KPHeuris.

Effectiveness of branching strategies. We evaluate the effectiveness of branching rules by comparing our kPEX with its variants which use different branching strategies. In specific, the variant kPEX-b1 adopts the existing branching strategy of kPlexT (whose time complexity is $O^*((\delta d)^{k+1}\gamma_k^{\delta})$ [12]), while the variant kPEX-b2 uses the same branching strategy as in KPLEX (whose time complexity is $O^*((\delta d)^{k+1}(k+1)^{\delta+k-|S^*|})$ [53])[1]. The running times on 30 representative graphs with $k = 2$ and 5 are shown in Table 24. We can observe that none of the three branching strategies can constantly outperform the other two. For example, kPEX-b2 is the fastest among the three solvers on most instances when $k = 2$ but kPEX outperforms the others when $k = 5$. One possible reason could be that the polynomial factor, the base and the exponent of the exponential time complexity all contribute to the theoretical time complexity. These three parts are different for the three branching strategies given a specific graph, e.g., kPEX-b1 has a smaller base γ_k than kPEX-b2 while kPEX-b2 has a smaller exponent than kPEX-b1.

[1]The O^* notation hides polynomial factors; δ, d and $|S^*|$ denote the degeneracy, the maximum degree and the size of maximum k-plex of G; γ_k is only related to k and $\gamma_k < 2$.

Table 12: Running time in seconds of kPEX and state-of-the-arts on 30 graphs with $k = 15$

ID	kPEX (ours)	KPLEX	kPlexT	kPlexS	DiseMKP
G1	**7.79**	44.29	34.96	17.14	401.75
G2	**0.08**	34.53	3.65	8.40	22.43
G3	OOT	OOT	OOT	OOT	OOT
G4	OOT	OOT	OOT	OOT	OOT
G5	OOT	OOT	OOT	OOT	OOT
G6	OOT	OOT	OOT	OOT	OOT
G7	OOT	OOT	OOT	OOT	OOT
G8	OOT	OOT	OOT	OOT	OOT
G9	**112.81**	OOT	OOT	OOT	OOT
G10	OOT	OOT	OOT	OOT	OOT
G11	**17.98**	OOT	OOT	OOT	OOT
G12	**0.59**	1.08	0.70	1.22	OOT
G13	**1298.97**	OOT	OOT	OOT	OOT
G14	OOT	OOT	OOT	OOT	OOT
G15	**139.97**	OOT	OOT	OOT	OOT
G16	**1.72**	2.02	OOT	2.43	OOT
G17	**6.63**	14.37	OOT	321.37	OOT
G18	**0.16**	0.33	1127.83	0.37	2576.73
G19	**0.50**	4.43	3.11	4.52	OOT
G20	590.13	194.44	OOT	**143.16**	OOT
G21	**119.36**	OOT	OOT	OOT	OOT
G22	**1.76**	8.63	33.07	8.25	OOT
G23	13.16	132.50	**5.74**	11.09	OOT
G24	**280.16**	OOT	OOT	OOT	OOT
G25	**5.70**	168.11	OOT	OOT	OOT
G26	**1.09**	3.63	3.67	3.52	3.51
G27	**1.83**	11.92	OOT	10.16	OOT
G28	OOT	OOT	OOT	OOT	OOT
G29	**2.46**	4.08	18.92	4.25	34.55
G30	OOT	OOT	OOT	OOT	OOT

Table 13: Running time in seconds of kPEX and state-of-the-arts on 30 graphs with $k = 20$

ID	kPEX (ours)	KPLEX	kPlexT	kPlexS	DiseMKP
G1	**0.00**	0.00	0.00	0.00	**0.00**
G2	0.00	0.00	0.00	0.00	**0.00**
G3	OOT	OOT	OOT	OOT	OOT
G4	OOT	OOT	OOT	OOT	OOT
G5	OOT	OOT	OOT	OOT	OOT
G6	OOT	OOT	OOT	OOT	OOT
G7	OOT	OOT	OOT	OOT	OOT
G8	OOT	OOT	OOT	OOT	OOT
G9	**36.41**	OOT	OOT	OOT	OOT
G10	OOT	OOT	OOT	OOT	OOT
G11	**35.11**	OOT	OOT	OOT	OOT
G12	**0.38**	1.08	0.66	1.15	OOT
G13	**991.67**	OOT	OOT	OOT	OOT
G14	**2978.97**	OOT	OOT	OOT	OOT
G15	**396.27**	OOT	OOT	OOT	OOT
G16	**9.81**	311.28	OOT	2449.56	OOT
G17	46.48	**41.72**	OOT	1813.37	OOT
G18	**0.19**	0.83	OOT	0.77	OOT
G19	**0.55**	4.50	2.67	4.97	OOT
G20	111.26	1398.04	OOT	**54.02**	OOT
G21	**396.82**	OOT	OOT	OOT	OOT
G22	**1.85**	7.64	OOT	8.37	OOT
G23	12.89	130.41	**5.29**	29.54	OOT
G24	**1014.88**	OOT	OOT	OOT	OOT
G25	**10.31**	334.13	OOT	OOT	OOT
G26	**1.08**	3.80	3.39	3.43	2.68
G27	**2.24**	11.52	OOT	11.86	OOT
G28	OOT	OOT	OOT	OOT	OOT
G29	**2.46**	2.99	3.07	3.10	8.59
G30	OOT	OOT	OOT	OOT	OOT

Table 14: Running time in seconds of kPEX and its variants on 30 graphs with $k = 2$

ID	kPEX	kPEX-SeqRB	kPEX-CTCP	kPEX-EGo	kPEX-Degen
G1	1.11	1.23	**0.95**	0.95	0.95
G2	**1932.28**	2021.27	1988.00	1992.98	1994.67
G3	**50.47**	75.39	51.66	52.01	52.18
G4	1.47	2.90	**1.34**	1.36	1.36
G5	**74.91**	118.45	77.47	77.41	77.25
G6	OOT	OOT	OOT	OOT	OOT
G7	OOT	OOT	OOT	OOT	OOT
G8	0.33	0.29	0.33	**0.29**	0.31
G9	**38.64**	50.32	39.81	39.97	41.21
G10	4.17	5.16	4.05	**3.81**	3.88
G11	**20.57**	33.06	21.47	23.02	25.65
G12	0.69	**0.55**	2.41	0.96	1.12
G13	**279.91**	407.48	291.83	345.93	343.31
G14	**3082.80**	OOT	3186.76	OOT	OOT
G15	**138.28**	189.20	158.75	172.91	277.23
G16	2.05	**2.00**	6.23	3.34	3.33
G17	**6.30**	7.06	13.16	9.55	7.25
G18	2.40	2.50	2.89	**2.39**	2.56
G19	**8.81**	8.85	33.79	16.26	OOT
G20	5.62	**4.91**	245.87	190.31	316.79
G21	**13.08**	22.35	13.87	20.14	19.76
G22	**3.18**	3.20	22.43	11.40	3.43
G23	8.61	**8.11**	1418.09	469.20	8.54
G24	**24.07**	41.83	26.23	24.27	36.39
G25	**13.39**	17.56	86.75	35.87	13.80
G26	11.38	21.19	**11.03**	12.31	11.56
G27	3.35	**3.31**	30.24	13.21	3.58
G28	139.40	**137.51**	OOT	OOT	OOT
G29	**5.92**	7.65	6.46	10.13	10.28
G30	**136.74**	179.48	1453.76	1437.70	251.08

Table 15: Running time in seconds of kPEX and its variants on 30 graphs with $k = 3$

ID	kPEX	kPEX-SeqRB	kPEX-CTCP	kPEX-EGo	kPEX-Degen
G1	**5.25**	16.99	5.40	5.41	5.42
G2	OOT	OOT	OOT	OOT	OOT
G3	**58.89**	530.53	60.48	60.65	60.88
G4	**9.08**	59.21	9.20	9.53	9.52
G5	0.14	0.11	0.10	0.10	**0.09**
G6	2.75	21.44	**2.62**	2.85	3.18
G7	OOT	OOT	OOT	OOT	OOT
G8	0.21	0.18	**0.15**	0.17	0.17
G9	**20.59**	149.41	21.22	25.29	28.53
G10	**14.66**	82.46	15.03	14.80	15.20
G11	**6.53**	33.43	7.15	11.34	12.64
G12	0.53	**0.41**	2.29	0.92	0.91
G13	**158.38**	985.84	167.20	195.70	208.35
G14	OOT	OOT	OOT	OOT	OOT
G15	**45.68**	204.47	62.51	63.44	57.47
G16	**1.58**	1.68	5.91	2.26	1.87
G17	**2.34**	3.31	8.94	4.53	2.76
G18	**1.88**	3.96	2.30	2.36	2.49
G19	1.17	**1.00**	24.22	8.57	49.43
G20	**4.24**	4.40	226.14	171.87	271.25
G21	**9.00**	48.31	9.66	9.64	9.53
G22	2.88	**2.81**	20.21	9.99	3.34
G23	**12.68**	12.77	1617.71	468.21	118.89
G24	**14.90**	80.36	16.67	16.34	16.61
G25	**6.87**	13.36	76.74	28.93	7.86
G26	6.85	7.14	**6.34**	7.57	6.75
G27	**2.87**	2.88	27.60	12.35	3.18
G28	**109.15**	116.05	OOT	OOT	3271.82
G29	**112.05**	455.48	115.38	113.24	118.54
G30	**189.61**	833.98	1502.80	1356.16	353.15

Table 16: Running time in seconds of kPEX and its variants on 30 graphs with $k = 10$

ID	kPEX	kPEX-SeqRB	kPEX-CTCP	kPEX-EGo	kPEX-Degen
G1	**450.15**	1238.20	463.16	464.53	464.13
G2	**37.72**	50.23	38.79	38.71	38.58
G3	OOT	OOT	OOT	OOT	OOT
G4	OOT	OOT	OOT	OOT	OOT
G5	**14.24**	912.36	14.65	14.72	14.67
G6	OOT	OOT	OOT	OOT	OOT
G7	OOT	OOT	OOT	OOT	OOT
G8	OOT	OOT	OOT	OOT	OOT
G9	**301.28**	837.83	310.32	625.51	623.68
G10	OOT	OOT	OOT	OOT	OOT
G11	**6.01**	26.39	6.66	17.61	17.30
G12	0.65	**0.52**	2.22	0.83	0.60
G13	**835.65**	OOT	866.51	1206.11	1200.79
G14	**3110.98**	OOT	3207.34	3366.27	3346.56
G15	**22.56**	234.55	35.02	35.29	31.29
G16	**2.11**	3.71	6.10	4.78	4.75
G17	**1.79**	13.22	7.91	6.87	5.07
G18	**0.49**	21.17	1.09	3.77	3.57
G19	0.36	**0.29**	2.81	44.58	OOT
G20	**4.90**	7.01	204.72	108.26	225.67
G21	**20.44**	125.24	21.52	50.77	50.31
G22	2.09	**1.97**	15.59	7.08	2.05
G23	**12.59**	12.81	1613.00	462.58	116.71
G24	**33.64**	195.37	35.84	52.85	51.96
G25	2.87	**2.73**	58.16	17.78	2.91
G26	1.18	**1.09**	2.87	2.55	1.48
G27	**2.33**	2.33	23.40	9.59	2.38
G28	**260.74**	261.12	OOT	OOT	OOT
G29	2.82	3.18	2.86	2.76	**2.58**
G30	OOT	OOT	OOT	OOT	OOT

Table 17: Running time in seconds of kPEX and its variants on 30 graphs with $k = 15$

ID	kPEX	kPEX-SeqRB	kPEX-CTCP	kPEX-EGo	kPEX-Degen
G1	7.79	10.63	7.85	**7.72**	7.94
G2	0.08	0.06	0.05	0.06	**0.05**
G3	OOT	OOT	OOT	OOT	OOT
G4	OOT	OOT	OOT	OOT	OOT
G5	OOT	OOT	OOT	OOT	OOT
G6	OOT	OOT	OOT	OOT	OOT
G7	OOT	OOT	OOT	OOT	OOT
G8	OOT	OOT	OOT	OOT	OOT
G9	**112.81**	120.17	115.87	113.30	118.43
G10	OOT	OOT	OOT	OOT	OOT
G11	**17.98**	19.24	18.37	71.93	74.15
G12	0.59	**0.47**	2.11	0.78	0.62
G13	**1298.97**	1497.62	1329.66	2796.92	2867.16
G14	OOT	OOT	OOT	OOT	OOT
G15	**139.97**	167.43	148.50	170.58	173.08
G16	**1.72**	1.86	5.55	2.48	2.98
G17	6.63	7.52	12.19	7.16	**5.70**
G18	0.16	**0.13**	0.67	0.34	0.18
G19	0.50	**0.40**	0.70	728.27	OOT
G20	**590.13**	1240.20	785.65	943.65	1486.61
G21	**119.36**	131.19	121.98	225.87	233.15
G22	1.76	1.68	12.05	5.75	**1.66**
G23	13.16	**13.15**	1605.10	458.65	116.44
G24	**280.16**	316.40	286.53	281.00	293.41
G25	**5.70**	5.74	55.64	19.29	5.72
G26	**1.09**	1.10	2.72	2.49	1.48
G27	1.83	**1.75**	17.87	7.54	1.81
G28	OOT	OOT	OOT	OOT	OOT
G29	2.46	2.57	2.54	2.15	**2.10**
G30	OOT	OOT	OOT	OOT	OOT

Table 18: Running time in seconds of kPEX and its variants on 30 graphs with $k = 20$

ID	kPEX	kPEX-SeqRB	kPEX-CTCP	kPEX-EGo	kPEX-Degen
G1	**0.00**	0.00	**0.00**	0.00	**0.00**
G2	0.00	0.00	0.00	**0.00**	**0.00**
G3	OOT	OOT	OOT	OOT	OOT
G4	OOT	OOT	OOT	OOT	OOT
G5	OOT	OOT	OOT	OOT	OOT
G6	OOT	OOT	OOT	OOT	OOT
G7	OOT	OOT	OOT	OOT	OOT
G8	OOT	OOT	OOT	OOT	OOT
G9	36.41	**35.42**	35.87	71.18	74.99
G10	OOT	OOT	OOT	OOT	OOT
G11	**35.11**	35.96	35.82	58.23	59.62
G12	0.38	**0.30**	1.88	0.81	0.56
G13	**991.67**	1010.12	1020.25	OOT	OOT
G14	**2978.97**	3450.98	3055.50	OOT	OOT
G15	**396.27**	416.74	412.44	1627.16	1688.80
G16	9.81	10.78	10.71	**9.37**	24.72
G17	**46.48**	58.52	52.69	48.09	49.33
G18	**0.19**	0.19	0.67	0.35	0.21
G19	0.55	**0.45**	0.57	OOT	OOT
G20	**111.26**	122.56	274.94	625.36	OOT
G21	**396.82**	408.68	407.72	572.13	600.41
G22	1.85	**1.76**	12.48	5.96	2.10
G23	12.89	**12.65**	1602.86	452.09	116.28
G24	**1014.88**	1060.97	1040.63	1134.76	1191.44
G25	**10.31**	10.50	58.15	24.76	11.29
G26	**1.08**	1.10	2.36	2.34	1.39
G27	2.24	**2.10**	16.94	7.54	2.17
G28	OOT	OOT	OOT	OOT	OOT
G29	2.46	2.47	2.45	1.91	**1.84**
G30	OOT	OOT	OOT	OOT	**878.37**

Table 19: Pre-processing time in seconds on 20 graphs with k=2 (lb denotes the size of the computed heuristic k-plex)

ID	kPEX		kPlexT		kPlexS		DiseMKP	
	time	lb	time	lb	time	lb	time	lb
G11	1.22	**37**	0.44	**37**	**0.33**	34	0.70	35
G12	**0.66**	**63**	0.82	62	1.10	21	1.91	23
G13	8.89	**26**	**1.61**	25	2.14	25	3.77	23
G14	24.57	**54**	4.67	52	**2.71**	51	6.41	52
G15	7.68	**36**	**2.72**	**36**	3.33	32	13.19	34
G16	1.27	**19**	**0.90**	17	1.67	9	4.12	9
G17	1.32	**28**	**0.82**	27	1.62	27	6.13	26
G18	0.42	**70**	0.52	**70**	**0.38**	69	0.70	**70**
G19	4.70	**35**	1.03	34	**0.83**	30	15.40	31
G20	3.97	**15**	**3.34**	14	7.11	10	127.78	4
G21	1.16	**32**	**0.50**	30	0.52	30	0.84	28
G22	**3.18**	**31**	4.34	**31**	17.15	17	20.52	18
G23	8.61	**6**	**3.93**	5	8.74	5	992.05	5
G24	1.88	**32**	**0.87**	**32**	0.95	30	1.60	30
G25	**4.14**	**60**	7.54	**60**	29.69	**60**	56.94	59
G26	3.16	**872**	**3.08**	**872**	3.31	**872**	3.12	**872**
G27	**3.30**	**27**	4.31	**27**	15.77	24	24.84	24
G28	97.71	**11**	**54.90**	10	153.69	9	OOT	-
G29	2.68	273	2.79	**274**	**2.65**	271	9.48	272
G30	**99.54**	**52**	112.54	51	240.49	10	1768.07	10

Table 20: Pre-processing time in seconds on 20 graphs with k=3 (lb denotes the size of the computed heuristic k-plex)

ID	`kPEX`		`kPlexT`		`kPlexS`		`DiseMKP`	
	time	lb	**time**	lb	**time**	lb	**time**	lb
G11	0.84	**44**	0.55	42	**0.35**	40	0.72	41
G12	**0.52**	**67**	0.76	66	1.17	26	1.87	27
G13	7.46	**30**	**1.76**	29	1.91	28	4.02	27
G14	11.45	**58**	4.53	**58**	**2.80**	56	6.32	56
G15	4.56	**41**	**2.11**	40	3.75	40	12.24	39
G16	1.24	**22**	**0.79**	21	1.66	11	4.03	10
G17	1.07	**33**	**0.80**	32	1.78	31	5.95	29
G18	0.53	**77**	0.49	75	**0.35**	74	0.65	74
G19	1.16	**39**	0.95	**39**	**0.73**	34	14.34	34
G20	**3.38**	**17**	4.09	16	6.72	10	125.75	6
G21	1.13	**35**	0.54	34	**0.52**	33	0.85	33
G22	**2.88**	**32**	3.94	31	14.14	20	20.34	18
G23	12.68	**7**	**4.83**	**7**	10.65	6	1620.72	6
G24	1.78	**35**	**0.90**	34	1.12	31	1.70	31
G25	**3.86**	**66**	7.31	**66**	28.58	64	52.90	65
G26	3.15	**875**	2.95	**875**	**2.80**	**875**	3.07	**875**
G27	**2.86**	**32**	4.30	**32**	14.75	27	22.88	28
G28	**70.56**	**13**	132.97	12	255.37	9	OOT	-
G29	**2.46**	**280**	2.76	**280**	2.61	**280**	8.46	279
G30	110.38	**57**	**107.75**	55	204.71	12	1436.05	12

Table 21: Pre-processing time in seconds on 20 graphs with k=10 (lb denotes the size of the computed heuristic k-plex)

ID	`kPEX`		`kPlexT`		`kPlexS`		`DiseMKP`	
	time	lb	**time**	lb	**time**	lb	**time**	lb
G11	**0.33**	**67**	0.45	65	0.36	65	0.65	**67**
G12	**0.65**	**82**	0.73	74	1.11	45	1.97	45
G13	2.47	**54**	**1.38**	52	2.50	52	4.09	**54**
G14	4.62	**90**	4.33	88	**3.48**	88	6.23	87
G15	**1.70**	**65**	1.80	64	3.33	64	8.96	62
G16	0.86	**40**	**0.82**	36	1.24	25	3.86	24
G17	**0.49**	**53**	0.71	48	1.71	48	5.18	49
G18	**0.09**	**102**	0.36	101	0.30	101	0.59	101
G19	**0.35**	**53**	5.21	47	1.31	44	1.73	44
G20	**3.76**	**31**	31.92	29	25.36	20	76.75	20
G21	0.69	**57**	0.67	54	**0.51**	54	0.61	54
G22	**2.09**	**44**	3.13	38	8.87	34	12.11	35
G23	12.59	**21**	**4.79**	**21**	10.56	20	1581.10	20
G24	1.03	**57**	**0.69**	55	0.88	55	1.26	54
G25	**2.50**	**92**	5.54	91	20.98	91	40.37	91
G26	**1.18**	**891**	3.74	**891**	3.68	**891**	3.84	**891**
G27	**2.33**	**46**	4.03	45	11.30	40	18.89	41
G28	**239.64**	**27**	1723.92	22	3279.80	21	3424.16	21
G29	**2.39**	**316**	2.76	**316**	2.80	**316**	7.98	315
G30	**95.22**	**82**	95.33	77	201.61	26	875.39	26

Table 22: Pre-processing time in seconds on 20 graphs with k=15 (lb denotes the size of the computed heuristic k-plex)

ID	`kPEX`		`kPlexT`		`kPlexS`		`DiseMKP`	
	time	lb	**time**	lb	**time**	lb	**time**	lb
G11	0.26	**79**	0.42	77	**0.26**	77	0.59	77
G12	**0.59**	**89**	0.70	78	1.14	54	1.79	55
G13	1.96	**67**	**1.38**	63	2.07	63	3.76	65
G14	**2.62**	**108**	4.28	107	3.12	107	6.27	107
G15	1.43	**77**	**1.40**	76	3.38	76	7.87	74
G16	**0.83**	**51**	0.85	47	1.24	33	3.35	33
G17	**0.58**	**64**	0.66	59	1.63	59	4.95	57
G18	**0.07**	**116**	0.32	115	0.26	115	0.52	115
G19	**0.49**	**59**	2.30	50	1.50	49	0.51	49
G20	**4.20**	**41**	26.95	37	27.58	30	56.00	30
G21	0.58	**69**	0.63	67	**0.37**	67	0.75	68
G22	**1.76**	**49**	3.09	42	8.21	42	12.24	42
G23	13.16	**31**	**5.74**	**31**	10.49	30	1597.21	30
G24	0.81	**69**	**0.67**	**69**	0.85	**69**	1.31	66
G25	**2.32**	**101**	5.42	**101**	22.56	**101**	38.04	**101**
G26	**1.09**	**900**	3.67	**900**	3.52	**900**	3.51	**900**
G27	**1.82**	**53**	3.61	51	10.07	51	17.01	50
G28	**462.28**	**36**	1225.52	31	3033.48	31	2577.52	31
G29	**2.32**	**332**	2.88	**332**	3.67	**332**	8.42	331
G30	**78.45**	**98**	83.42	92	158.15	36	629.20	36

Table 23: Pre-processing time in seconds on 20 graphs with k=20 (lb denotes the size of the computed heuristic k-plex)

ID	`kPEX`		`kPlexT`		`kPlexS`		`DiseMKP`	
	time	lb	**time**	lb	**time**	lb	**time**	lb
G11	**0.22**	**89**	0.38	88	0.31	88	0.42	87
G12	**0.38**	**96**	0.66	78	1.08	64	1.57	63
G13	2.48	**79**	**1.36**	76	2.19	76	3.91	77
G14	**2.28**	**123**	3.94	122	3.03	122	6.02	121
G15	1.71	**88**	**1.67**	85	3.12	85	6.79	86
G16	3.07	**59**	1.28	51	**0.79**	40	1.55	40
G17	**0.58**	**74**	0.62	67	1.58	67	4.62	67
G18	**0.06**	**124**	0.30	123	0.19	123	0.49	123
G19	0.41	**64**	1.08	54	1.13	54	**0.36**	54
G20	**5.42**	**50**	29.64	44	19.73	40	43.92	40
G21	0.76	**79**	0.59	76	**0.45**	76	0.73	78
G22	**1.84**	**55**	3.48	47	8.27	47	10.91	48
G23	12.89	**41**	**5.29**	**41**	10.88	40	1637.07	40
G24	0.92	**78**	**0.67**	77	0.83	77	1.23	75
G25	**2.23**	**111**	5.25	110	20.05	110	37.19	110
G26	**1.08**	**910**	3.39	**910**	3.43	**910**	2.68	**910**
G27	**1.91**	**60**	3.13	58	10.73	58	15.40	58
G28	1875.37	**45**	**725.06**	40	2004.11	40	901.23	40
G29	**2.37**	**343**	3.04	**343**	2.98	**343**	7.97	342
G30	**76.53**	**112**	76.83	104	142.27	46	529.06	45

Table 24: Running time in seconds of variants of kPEX using different branching strategies on 30 representative graphs

ID	$k = 2$			$k = 5$		
	kPEX	**kPEX-b1**	**kPEX-b2**	**kPEX**	**kPEX-b1**	**kPEX-b2**
G1	1.11	1.39	**0.96**	**4.02**	23.6	6.67
G2	1932.28	2255.77	**850.64**	**1324.85**	OOT	OOT
G3	50.47	64.27	**29.33**	**1485.70**	OOT	3095.79
G4	1.47	1.9	**1.44**	**297.49**	589.22	320.8
G5	74.91	76.96	**2.36**	**0.14**	0.45	0.14
G6	OOT	OOT	**1576.69**	**24.11**	OOT	302.7
G7	OOT	OOT	OOT	**421.54**	OOT	OOT
G8	**0.33**	0.61	0.42	**2.17**	3.84	2.4
G9	38.64	49.96	**15.77**	**2.75**	481.56	5.7
G10	4.17	5.87	**3.75**	**186.24**	337.9	196.41
G11	20.57	26.66	**13.89**	**4.02**	90.34	5.79
G12	**0.69**	0.7	0.69	0.49	**0.38**	0.49
G13	279.91	432.28	**220.51**	**53.98**	1017.53	69.92
G14	3082.8	OOT	**1209.01**	**901.50**	OOT	2693.07
G15	138.28	203.5	**108.77**	**20.89**	304.76	25.24
G16	2.05	2.38	**1.75**	1.81	2.69	**1.70**
G17	6.3	7.97	**5.26**	**1.63**	6.06	1.98
G18	2.4	2.61	**1.92**	**0.86**	2.81	1.26
G19	8.81	**8.60**	8.82	0.82	**0.77**	0.77
G20	5.62	6.23	**5.31**	4.55	4.37	**4.27**
G21	13.08	17.66	**8.92**	**3.00**	26.14	3.01
G22	3.18	**3.15**	3.22	2.43	2.33	**2.31**
G23	8.61	8.73	**8.11**	12.89	12.78	**12.52**
G24	24.07	33.33	**16.61**	**5.56**	44.68	6.14
G25	13.39	16.08	**9.20**	3.63	3.69	**3.54**
G26	11.38	11.54	**10.53**	4.87	**4.81**	4.85
G27	3.35	3.37	**3.16**	**2.43**	2.52	2.58
G28	139.4	144.24	**138.37**	132.41	**130.81**	133.18
G29	5.92	5.9	**4.66**	**6.01**	91.44	7.71
G30	136.74	139.27	**119.29**	**589.81**	OOT	1715.71